

Learning in Clausal Logic: A Perspective on Inductive Logic Programming

Peter Flach¹ and Nada Lavrač²

¹ University of Bristol
Woodland Road, Bristol BS8 1UB, United Kingdom
Peter.Flach@bristol.ac.uk, <http://www.cs.bris.ac.uk/~flach/>

² Jožef Stefan Institute
Jamova 39, 1000 Ljubljana, Slovenia
Nada.Lavrac@ijs.si, <http://www-ai.ijs.si/NadaLavrac/>

Abstract. Inductive logic programming is a form of machine learning from examples which employs the representation formalism of clausal logic. One of the earliest inductive logic programming systems was Ehud Shapiro's Model Inference System [90], which could synthesise simple recursive programs like `append/3`. Many of the techniques devised by Shapiro, such as top-down search of program clauses by refinement operators, the use of intensional background knowledge, and the capability of inducing recursive clauses, are still in use today. On the other hand, significant advances have been made regarding dealing with noisy data, efficient heuristic and stochastic search methods, the use of logical representations going beyond definite clauses, and restricting the search space by means of declarative bias. The latter is a general term denoting any form of restrictions on the syntactic form of possible hypotheses. These include the use of types, input/output mode declarations, and clause schemata. Recently, some researchers have started using alternatives to Prolog featuring strong typing and real functions, which alleviate the need for some of the above ad-hoc mechanisms. Others have gone beyond Prolog by investigating learning tasks in which the hypotheses are not definite clause programs, but for instance sets of indefinite clauses or denials, constraint logic programs, or clauses representing association rules. The chapter gives an accessible introduction to the above topics. In addition, it outlines the main current research directions which have been strongly influenced by recent developments in data mining and challenging real-life applications.

1 Introduction

Inductive logic programming has its roots in concept learning from examples, a relatively straightforward form of induction that has been studied extensively by machine learning researchers [70]. The aim of concept learning is to discover, from a given set of pre-classified examples, one or several classification rules with high predictive power. For many concept learning tasks, so-called attribute-value

languages have sufficient representational power. An example of an attribute-value classification rule is the following (regarding contact lens prescriptions, from [97]):

*IF Age = Pre-presbyopic AND Astigmatic = No AND Tear-production = Normal
THEN Recommendation = Soft*

A learned concept definition could consist of several of such rules. Concept learning can be generalised to multi-class classification problems, where one would learn a set of rules for each class. (In contrast, in concept learning we are usually not interested in learning rules for the complement of the concept.)

When objects are structured and consist of several related parts, we need a richer representation formalism with variables to refer to those parts. In the 1970s and '80s machine learning researchers started exploring the use of logic programming representations, which led to the establishment of inductive logic programming (ILP) [79] as a subdiscipline at the intersection of machine learning and computational logic. Recent years have seen a steady increase in ILP research, as well as numerous applications to practical problems like data mining and scientific discovery – see [8,30] for an overview of such applications.

Most of the current real-world ILP applications involve predictive knowledge discovery, in particular the induction of classification and prediction rules from a given database of examples and the available background knowledge. Successful ILP applications include drug design [55], protein secondary structure prediction [78], mutagenicity prediction [93], carcinogenesis prediction [94], medical diagnosis [72], discovery of qualitative models in medicine [48], finite-element mesh design [28], telecommunications [92], natural language processing [73], recovering software specifications [12], and many others.

Focusing on problems such as data mining also led away from pure classification problems, where a teacher would pre-classify the training data and the learning problem consists in coming up with rules predicting the class. In descriptive induction problems, there is no notion of a class and the goal of learning is to come up with rules describing correlations between any descriptors found in the data. A typical example here are association rules [1], which are useful in applications like market basket analysis. Descriptive induction methods have also been studied in a first-order context.

The outline of the chapter is as follows. Sections 2 and 3 introduce the tasks of predictive and descriptive ILP, respectively. In Section 4 we take a closer look at the different knowledge formalisms that are used in ILP, in particular Datalog, Prolog, typed and functional logic languages, and database representations. Section 5 overviews state-of-the-art ILP techniques and systems, and in Section 6 we look at future challenges for ILP research and applications, including a section on research challenges related to computational logic. Section 7 concludes.

2 Predictive ILP

In this section we give a tutorial introduction to the main forms of predictive inductive logic programming. One instance of a predictive ILP problem concerns the inductive construction of an intensional predicate definition (a set of Horn clauses with a single predicate in the head) from a selection of ground instances of the predicate. More generally, there can be several predicates whose definitions are to be learned, also called *foreground* predicates or *observables*. In the general case, this requires suitably defined auxiliary or *background predicates* (simple recursive predicates such as `member/2` and `append/3` notwithstanding). The induced set of rules or *inductive hypothesis* then provides an intensional connection between the foreground predicates and the background predicates; we will sometimes call such rules *foreground* rules. We will also use the terms *facts* to refer to extensional knowledge, and *rules* to refer to intensional knowledge. The terms ‘knowledge’ or ‘theory’ may refer to both facts and rules. Thus, predictive induction infers foreground rules from foreground facts and background theory.

Definition 1 (Predictive ILP). *Let P_F and N_F be sets of ground facts over a set of foreground predicates F , called the positive examples and the negative examples, respectively. Let T_B , the background theory, be a set of clauses over a set of background predicates B . Let L be a language bias specifying a hypothesis language \mathcal{H}_L over $F \cup B$ (i.e., a set of clauses). A predictive ILP task consists in finding a hypothesis $H \subseteq \mathcal{H}_L$ such that $\forall p \in P_F : T_B \cup H \models p$ and $\forall n \in N_F : T_B \cup H \not\models n$.*

The subscripts F and B are often dropped, if the foreground and background predicates are understood. We will sometimes refer to all examples collectively as E .

Definition 1 is under-specified in a number of ways. First, it doesn’t rule out trivial solutions like $H = P$ unless this is excluded by the language bias (which is not often the case since the language bias cannot simply exclude ground facts, because they are required by certain recursive predicate definitions). Furthermore, the definition doesn’t capture the requirement that the inductive hypothesis correctly predicts unseen examples. It should therefore be seen as a general framework, which needs to be further instantiated to capture the kinds of ILP tasks addressed in practice. We proceed by briefly discussing a number of possible variations, indicating which of these we can handle with the approach proposed in this chapter.

Clauses in T and H are often restricted to definite clauses with only positive literals in the body. Some ILP algorithms are able to deal with normal clauses which allow negative literals in the body. One can go a step further and allow negation over several related literals in the body (called *features* in [65]).

In a typical predictive ILP task, there is a single foreground predicate to be learned, often referred to as the *target predicate*. In contrast, *multiple predicate learning* occurs when $|F| > 1$. Multiple predicate learning is hard if the foreground predicates are mutually dependent, i.e., if one foreground predicate

acts as an auxiliary predicate to another foreground predicate, because in that case the auxiliary predicate is incompletely specified. Approaches to dealing with incomplete background theory, such as abductive concept learning [52], can be helpful here. Alternatively, multiple predicate learning may be more naturally handled by a descriptive ILP approach, which is not intended at learning of classification rules but at learning of properties or constraints that hold for E given T (see Section 3). The problems of learning recursive rules, where a foreground predicate is its own auxiliary predicate, are related to the problems of multiple predicate learning.

Definition 1 only applies to boolean classification problems. The definition could be extended to multi-class problems, by supplying the foreground predicate with an extra argument indicating the class. In such a case, a set of rules has to be learned for each class. It follows that we can also distinguish binary classification problems in which both the positive and negative class have to be learned explicitly (rather than by negation-as-failure, as in the definition).

In *individual-centred* domains there is a notion of individual, e.g. molecules or trains, and learning occurs on the level of individuals only. Usually, individuals are represented by a single variable, and the foreground predicates are either unary predicates concerning boolean properties of individuals, or binary predicates assigning an attribute-value or a class-value to each individual. Local variables referring to parts of individuals are introduced by so-called structural predicates. Individual-centred representations allow for a strong language bias for feature construction (see Section 4.2). On the other hand, most program synthesis tasks lack a clear notion of individual. Consider, for instance, the definition of *reverse/2*: if lists are seen as individuals – which seems most natural – the clauses are not classification rules; if pairs of lists are seen as individuals, turning the clauses into boolean classification rules, the learning system will have to rediscover the fact that the output list is determined by the input list.

Sometimes a predictive ILP task is unsolvable with the given background theory, but solvable if an additional background predicate is introduced. For instance, in Peano arithmetic multiplication is not finitely axiomatisable unless the definition of addition is available. The process of introducing additional background predicates during learning is called *predicate invention*. Predicate invention can also be seen as an extreme form of multiple predicate learning where some of the foreground predicates have no examples at all.

An initial foreground H_0 may be given to the learner as a starting point for hypothesis construction. Such a situation occurs e.g., in incremental learning, where examples become available one-by-one and are processed sequentially. Equivalently, we can perceive this as a situation where the background theory also partially defines the foreground predicate(s). This is usually referred to as *theory revision*.

After having considered the general form that predictive ILP problems may take, we now turn our attention to predictive ILP algorithms. Broadly speaking, there are two approaches. One can either start from short clauses, progressively adding literals to their bodies as long as they are found to be overly general (*top-*

down approaches); or one can start from long clauses, progressively removing literals until they would become overly general (*bottom-up* approaches). Below, we illustrate the main ideas by means of some simplified examples.

2.1 Top-Down Induction

Basically, top-down induction is a generate-then-test approach. Hypothesis clauses are generated in a pre-determined order, and then tested against the examples. Here is an example run of a fictitious incremental top-down ILP system:

<i>example</i>	<i>action</i>	<i>clause</i>
+m(a, [a, b])	add clause	m(X, Y)
-m(x, [a, b])	specialise:	try m(X, [])
		try m(X, [V W])
		try m(X, [X W])
+m(b, [b])	do nothing	
+m(b, [a, b])	add clause:	try m(X, [V W])
		try...
		try m(X, [V W]) :- m(X, W)

The hypothesis is initialised with the most general definition of the target predicate. After seeing the first negative example, this clause is specialised by constraining the second argument. Several possibilities have to be tried before we stumble upon a clause that covers the positive example but not the negative one. Fortunately, the second positive example is also covered by this clause. A third positive example however shows that the definition is still incomplete, which means that a new clause has to be added. The system may find such a clause by returning to a previously refuted clause and specialise it in a different way, in this case by adding a literal to its body.

The resulting clause being recursive, testing it against the examples means querying the predicate to be learned. Since in our example the base case had been found already this doesn't pose any problem; however, this requires that the recursive clause is learned last, which is not always under control of the teacher. Moreover, if the recursive clause that is being tested is incorrect, such as $m(X, Y) :- m(Y, X)$, this may lead to non-termination problems. An alternative approach, known as extensional coverage, is to query the predicate to be learned against the examples. Notice that this approach would succeed here as well because of the second positive example.

The approach illustrated here is basically that of Shapiro's *Model Inference System* [90,91], an ILP system *avant la lettre* (the term 'inductive logic programming' was coined in 1991 by Muggleton [75]). MIS is an incremental top-down system that performs a complete breadth-first search of the space of possible clauses. Shapiro called his specialisation operator a *refinement operator*, a term

that is still in use today (see [59] for an extensive analysis of refinement operators). A much simplified Prolog implementation of MIS can be found in [36]. Another well-known top-down system is Quinlan's Foil [86].

As the previous example shows, clauses can be specialised in two ways: by applying a substitution, and by adding a body literal. This is formalised by the relation of θ -subsumption, which establishes a syntactic notion of generality.

Definition 2 (θ -subsumption). *A clause C_1 θ -subsumes a clause C_2 iff there is a substitution θ such that all literals in $C_1\theta$ occur in C_2 .¹*

θ -subsumption is reflexive and transitive, but not antisymmetric (e.g., $\mathbf{p(X)} : \neg\mathbf{q(X)}$ and $\mathbf{p(X)} : \neg\mathbf{q(X)}, \mathbf{q(Y)}$ θ -subsume each other). It thus defines a pre-order on the set of clauses, i.e., a partially ordered set of equivalence classes. If we define a clause to be *reduced* if it does not θ -subsume any of its subclauses, then every equivalence class contains a reduced clause that is unique up to variable renaming. The set of these equivalence classes forms a lattice, i.e., two clauses have a unique least upper bound and greatest lower bound under θ -subsumption. We will refer to the least upper bound of two clauses under θ -subsumption as their θ -LGG (least general generalisation under θ -subsumption). Note that the lattice does contain infinite descending chains.

Clearly, if C_1 θ -subsumes C_2 then C_1 entails C_2 , but the reverse is not true. For instance, consider the following clauses:

```

nat(s(X)) : -nat(X) .
nat(s(s(Y))) : -nat(Y) .
nat(s(s(Z))) : -nat(s(Z)) .

```

Every model of the first clause is necessarily a model of the other two, both of which are therefore entailed by the first. However, the first clause θ -subsumes the third (substitute $\mathbf{s(Z)}$ for \mathbf{X}) but not the second. Gottlob characterises the distinction between θ -subsumption and entailment [47]: basically, C_1 θ -subsumes C_2 without entailing it if the resolution proof of C_2 from C_1 requires to use C_1 more than once.

It seems that the entailment ordering is the one to use, in particular when learning recursive clauses. Unfortunately, the least upper bound of two Horn clauses under entailment is not necessarily unique. The reason is simply that, generally speaking, this least upper bound would be given by the disjunction of the two clauses, but this may not be a Horn clause. Furthermore, generalisations under entailment are not easily calculated, whereas generalisation and specialisation under θ -subsumption are simple syntactic operations. Finally, entailment between clauses is undecidable, whereas θ -subsumption is decidable (but NP-complete). For these reasons, ILP systems usually employ θ -subsumption rather than entailment. Idestam-Almquist defines a stronger form of entailment called T-implication, which remedies some of the shortcomings of entailment [50,51].

¹ This definition, and the term θ -subsumption, was introduced in the context of induction by Plotkin [83,84]. In theorem proving the above version is termed subsumption, whereas θ -subsumption indicates a special case in which the number of literals of the subsumant does not exceed the number of literals of the subsumee [68].

2.2 Bottom-Up Induction

While top-down approaches successively specialise a very general starting clause, bottom-up approaches generalise a very specific bottom clause. Again we illustrate the main ideas by means of a simple example. Consider the following four ground facts:

$$\begin{array}{ll} a([1,2], [3,4], [1,2,3,4]) & a([2], [3,4], [2,3,4]) \\ a([a], [], [a]) & a([], [], []) \end{array}$$

Upon inspection we may conjecture that these ground facts are pairwise related by one recursion step, i.e., the following two clauses may be ground instances of the recursive clause in the definition of $a/3$:

$$\begin{array}{l} a([1,2], [3,4], [1,2,3,4]) :- \\ \quad a([2], [3,4], [2,3,4]) \\ a([a], [], [a]) :- \\ \quad a([], [], []) \end{array}$$

All that remains to be done is to construct the θ -LGG of these two ground clauses, which in this simple case can be constructed by anti-unification. This is the dual of unification, comparing subterms at the same position and turning them into a variable if they differ. To ensure that the resulting inverse substitution is the least general anti-unifier, we only introduce a new variable if the pair of different subterms has not been encountered before. We obtain the following result:

$$\begin{array}{l} a([A|B], C, [A|D]) :- \\ \quad a(B, C, D) \end{array}$$

which is easily recognised as the recursive clause in the standard definition of $append/3$.

In general things are of course much less simple. One of the main problems is to select the right ground literals from a much larger set. Suppose now that we know which head literals to choose, but not which body literals. One approach is to simply lump all literals together in the bodies of both ground clauses:

$$\begin{array}{l} a([1,2], [3,4], [1,2,3,4]) :- \\ \quad a([1,2], [3,4], [1,2,3,4]), a([a], [], [a]), \\ \quad a([], [], []), a([2], [3,4], [2,3,4]) \\ \\ a([a], [], [a]) :- \\ \quad a([1,2], [3,4], [1,2,3,4]), a([a], [], [a]), \\ \quad a([], [], []), a([2], [3,4], [2,3,4]) \end{array}$$

Since bodies of clauses are, logically speaking, unordered, the θ -LGG is obtained by anti-unifying all possible pairs of body literals, keeping in mind the variables that were introduced when anti-unifying the heads. Thus, the body of the resulting clause consists of 16 literals:

$$\begin{aligned}
& a([A|B], C, [A|D]) :- \\
& \quad a([1, 2], [3, 4], [1, 2, 3, 4]), a([A|B], C, [A|D]), \\
& \quad a(W, C, X), a([S|B], [3, 4], [S, T, U|V]), \\
& \quad a([R|G], K, [R|L]), a([a], [], [a]), \\
& \quad a(Q, [], Q), a([P], K, [P|K]), a(N, K, O), \\
& \quad a(M, [], M), a([], [], []), a(G, K, L), \\
& \quad a([F|G], [3, 4], [F, H, I|J]), a([E], C, [E|C]), \\
& \quad a(B, C, D), a([2], [3, 4], [2, 3, 4]).
\end{aligned}$$

After having constructed this bottom clause, our task is now to generalise it by throwing out as many literals as possible. To begin with, we can remove the ground literals, since they are our original examples. It also makes sense to remove the body literal that is identical to the head literal, since it turns the clause into a tautology. More substantially, it is reasonable to require that the clause is connected, i.e., that each body literal shares a variable with either the head or another body literal that is connected to the head. This allows us to remove another 7 literals, so that the clause becomes

$$\begin{aligned}
& a([A|B], C, [A|D]) :- \\
& \quad a(W, C, X), a([S|B], [3, 4], [S, T, U|V]), \\
& \quad a([E], C, [E|C]), a(B, C, D).
\end{aligned}$$

Until now we have not made use of any negative examples. They may now be used to test whether the clause becomes overly general, if some of its body literals are removed. Another, less crude way to get rid of body literals is to place restrictions upon the existential variables they introduce. For instance, we may require that they are determinate, i.e., have only one possible instantiation given an instantiation of the head variables and preceding determinate literals.

The approach illustrated here is essentially the one taken by Muggleton and Feng's Golem system [77] (again, a much simplified Prolog implementation can be found in [36]). Although Golem has been successfully applied to a range of practical problems, it has a few shortcomings. One serious restriction is that it requires ground background knowledge. Furthermore, all ground facts are lumped together, whereas it is generally possible to partition them according to the examples (e.g., the fact $a([a], [], [a])$ has clearly nothing to do with the fact $a([2], [3, 4], [2, 3, 4])$). Both restrictions are lifted in Muggleton's current ILP system Progol [80]. Essentially, Progol constructs a bottom clause for a selected example by adding its negation to the (non-ground) background theory and deriving all entailed negated body literals. By means of mode declarations (see Section 4.4) this clause is generalised as much as possible; the resulting body literals are then used in a top-down refinement search, guided by a heuristic which measures the amount of compression the clause achieves relative to the examples (see the next section on heuristics). Progol is thus a hybrid bottom-up/top-down system. It has been successfully applied to a number of scientific discovery problems.

The examples we used above to illustrate top-down and bottom-up ILP algorithms concerned inductive synthesis of simple recursive programs. While illus-

trative, these examples are non-typical of many ILP approaches which perform classification rather than program synthesis, use an individual-centred representation, and employ background knowledge rather than recursion. Examples of these kinds of ILP problems will be given in Section 4.

2.3 Heuristics

Shapiro's MIS searched the ordered set of hypothesis clauses in a breadth-first manner. Experience shows that this is too inefficient except for relatively restricted induction problems. In general every ILP system needs heuristics to direct the search. Heuristics are also needed if the data is noisy (contains errors). We can only scratch the surface of the topic here – for overviews see [63, Chapter 8] or [64].

There are basically three approaches to heuristics in machine learning. The statistical approach treats the examples as a sample drawn from a larger population. The (population) accuracy of a clause is the relative frequency of true instances among the instances covered by the clause (which is roughly the same as the number of substitutions that make body and head true divided by the number of substitutions that make the body true). Clearly, population accuracy is a number between 0 and 1, with 1 denoting perfect fit and 0 denoting total non-fit. As this is a population property it needs to be estimated from the sample. One obvious candidate is sample accuracy; when dealing with small samples corrections such as the Laplace estimate (which assumes a uniform prior distribution of the classes) or variations thereof can be applied. Informativity estimates are variants of accuracy estimates, which measure the entropy (impurity) of the set of examples covered by a clause with respect to their classification. One potential problem when doing best-first search is overfitting: if clauses are being specialised until they achieve perfect fit, they may cover only very few examples. To trade off accuracy and generality, the accuracy or informativity gain achieved by adding a literal to a clause is usually weighted with a fraction comparing the number of positive examples covered by each clause. In addition, ILP systems usually include a stopping criterion that is related to the estimated significance of the induced clause.

Bayesians do not treat probabilities as objective properties of an unknown sample, but rather as subjective degrees of belief that the learner is prepared to attach to a clause. The learner constantly updates these beliefs when new evidence comes in. This requires a *prior probability* distribution over the hypothesis space, which represents the degrees of belief the learner attaches to hypotheses in the absence of any evidence. It also requires conditional probability distributions over the example space for each possible hypothesis, which represents how likely examples are to occur given a particular hypothesis. The *posterior probability* of a hypothesis given the observed evidence, which is the heuristic we are going to maximise, is then calculated using Bayes' law. For instance, suppose that initially we consider a particular hypothesis to be very unlikely, but certain evidence to be very likely given that hypothesis. If subsequently we indeed observe that evidence, this will increase our belief that the hypothesis

might after all be true. One problem with the Bayesian approach is the large number of probability distributions that are required. Since they influence the posterior probability, they should be meaningful and justifiable. For instance, using a uniform prior distribution (all hypotheses are *a priori* equally likely) may be technically simple but hard to justify.

Finally, there is the compression approach [95]. The idea here is that the best hypothesis is the one which most compresses the data (for instance because the learner wants to transmit the examples over a communication channel in the most efficient way). One therefore compares the size of the examples with the size of the hypothesis. To measure these sizes one needs some form of encoding: for instance, if the language contains 10 predicate symbols one can assign each of them a number and encode this in binary in 4 bits (clearly the encoding should also be communicated but this is independent of the examples and the hypothesis). Similar to the Bayesian approach, this encoding needs to be justified: for instance, if variables are encoded in many bits and constants in few, there may be no non-ground hypothesis that compresses the data and generalisation will not occur.

In fact, there is a close link between the compression approach and the Bayesian approach as follows. Suppose one has to transmit one of n messages but does not know *a priori* which one. Suppose however that one does have a probability distribution over the n messages. Information theory tells us that the theoretically optimal code assigns $-\log_2 p_i$ bits to the i -th message (p_i is the probability of that message). Having thus established a link between a probability distribution and an encoding, we see that choosing an encoding in fact amounts to choosing a prior probability. The hypothesis with the highest posterior probability is the one which minimises the code length for the hypothesis plus the code length for the examples given the hypothesis (i.e., the correct classifications for those examples that are misclassified by the hypothesis). The compression approach and the Bayesian approach are really two sides of the same coin. One advantage of the compression viewpoint may be that encodings are conceptually simpler than distributions.

3 Descriptive ILP

Inductive logic programming started as an offspring of concept learning from examples, with attribute-value classification rules replaced by Prolog predicate definitions. As we have seen, this has naturally led to a definition of induction as inference of a target theory from some of its consequences and non-consequences (Definition 1). However, this definition assumes that the induced hypothesis will be used to derive further consequences. It is much less applicable to induce formulae with a different pragmatics, such as integrity constraints, for which we therefore need a different problem definition. The process of inducing non-classificatory rules is usually called *descriptive* induction, and if the representation formalism involved is clausal logic we may refer to it as descriptive ILP.

The fact that there is a pragmatic difference between intensional database rules and integrity constraints is common knowledge in the field of deductive databases, and the conceptual difference between inducing either of them is a natural one from that perspective. On the other hand, just as the topic of Horn logic is much better developed than the subject of integrity constraints, induction of the latter is a much more recent development than Horn clause induction, and some major research topics remain. For instance, giving up Horn logic means that we lose our main criterion for deciding whether a hypothesis is good or not: classification accuracy. It is not immediately obvious what task the induced constraints are going to perform. One important research problem is therefore to find meaningful heuristics for this kind of induction. Furthermore, it is hard to capture all forms of descriptive ILP in a single definition. The following definition therefore only provides a starting point for discussing different approaches to descriptive ILP.

Definition 3 (Descriptive ILP). *Let E be a collection of evidence and let m_E be a model constructed from E . Let L be a language bias specifying a hypothesis language \mathcal{H}_L . A descriptive ILP task consists in finding a hypothesis $H \subseteq \mathcal{H}_L$ axiomatising m_E , i.e., H is true in m_E , and $\forall g \in \mathcal{H}_L$: if g is true in m_E then $H \models g$.*

Definition 3 leaves the form of the evidence unspecified. In the simplest case, the evidence is simply an enumeration of the intended model m_E by ground facts. The evidence may also include an intensional part T , in which m_E would be the truth-minimal Herbrand model of $T \cup E$ (note that in descriptive ILP there is no real need to distinguish between intensional and extensional evidence, since they both end up at the same end of the turnstile). E could also be a collection of models, from which a canonical model m_E is constructed.

In general it can be said that, while predictive induction is driven by entailment, descriptive induction is driven by some notion of consistency or truth in a model. For instance, database constraints exclude certain database states, while intensional rules derive part of a database state from another part. In a sense, integrity constraints are learned by generalising from several database states. It is therefore often more natural to associate the extensional data with one or more models of the theory to be learned, rather than with a single ground atomic consequence. From this viewpoint induction of integrity constraints is more a descendant of one of the typical problems studied in computational learning theory, *viz.* learning arbitrary boolean expressions from some of its satisfying and falsifying assignments [96,3].

Furthermore, there is often a close link between descriptive induction and nonmonotonic or closed-world reasoning, in that both involve some form of Closed World Assumption (CWA). However, the inductive CWA has a slightly different interpretation: ‘everything I haven’t seen behaves like the things I have seen’ [49]. Sometimes this enables one to treat the data as specifying one preferred or minimal model, and develop the hypothesis from that starting point. Meta-logical properties of this form of inductive reasoning are therefore similar to those of reasoning with rules that tolerate exceptions [37,38].

Table 1. A feature table.

X	female(X)	male(X)	gorilla(X)
	-	-	-
	-	-	+
	-	+	-
richard, fred	-	+	+
	+	-	-
liz, ginger	+	-	+
	+	+	-
	+	+	+

We illustrate descriptive ILP with a simple example, taken from [21]. Let the evidence be given by the following ground facts:

```
gorilla(liz).      gorilla(richard).
gorilla(ginger).  gorilla(fred).
female(liz).      male(richard).
female(ginger).   male(fred).
```

One approach to construct a set of most general satisfied clauses is by means of DNF to CNF conversion [39]. From the evidence we construct a *feature table*, which is a sort of generalised truth-table (Table 1). We assume that a set of literals of interest has been generated in some way. Each column in the feature table corresponds to one of those literals, and each row corresponds to a grounding substitution of all variables in the literal set. In Table 1, the rows without an entry for X indicate that one cannot find a substitution for X such that the three ground atoms obtain the required truth value – these represent the so-called *countermodels*. For instance, the first line indicates that the evidence does not contain a substitution for X such that `female(X)`, `male(X)` and `gorilla(X)` are all false. The desired clausal theory can now be found by constructing the prime implicants of the countermodels and negating them. For instance, the first two countermodels together imply that $\neg \text{female}(X) \wedge \neg \text{male}(X)$ is unsatisfiable, i.e., `female(X);male(X)` is a most general satisfied clause. This yields the following theory:

```
gorilla(X).
male(X);female(X).
:-male(X),female(X).
```

Notice that a more specific hypothesis would be obtained by adding a non-female, non-male non-gorilla to the evidence (or by requiring that the evidence be range-restricted):

```
gorilla(X):-female(X)
gorilla(X):-male(X)
```

Table 2. A 3-dimensional contingency table.

	$\text{son}(X,Y)/5$	$\neg\text{son}(X,Y)/52$	$\text{son}(X,Y)$	$\neg\text{son}(X,Y)$
$\text{parent}(Y,X)/11$	0 (0.10)	6 (1.06)	5 (0.86)	0 (8.98)
$\neg\text{parent}(Y,X)/46$	0 (0.42)	0 (4.42)	0 (3.61)	46 (37.55)
	$\text{daughter}(X,Y)/6$		$\neg\text{daughter}(X,Y)/51$	

$\text{male}(X); \text{female}(X) : \neg\text{gorilla}(X)$
 $: \neg\text{male}(X), \text{female}(X)$

An important difference with the classification-oriented form of ILP is that here each clause can be discovered independently of the others. This means that the approach can be implemented as an any-time algorithm, at any time maintaining a hypothesis that is meaningful as an approximate solution, the sequence of hypotheses converging to the correct solution over time.

In Section 2.3, we discussed the use of heuristics in predictive ILP. The need for heuristics is even more urgent in descriptive ILP, because there are usually large numbers of rules satisfying the requirements (typically because the expensive condition that the rules are the most general ones is relaxed). We outline a possible approach inspired by [44].

Suppose we are considering the literals $\text{daughter}(X,Y)$, $\text{son}(X,Y)$, and $\text{parent}(Y,X)$. As in Table 1 we count the number of substitutions for each possible truthvalue assignment, but instead of a truthtable we employ a multi-dimensional contingency table to organise these counts (Table 2). This table contains the 8 cells of the 3-dimensional contingency table, as well as various marginal frequencies obtained by summing the relevant cells. Using these marginal frequencies we can now calculate expected frequencies for each cell under the assumption of independence of the three literals. For instance, the expected frequency of substitutions that make $\text{parent}(Y,X)$ true, $\text{daughter}(X,Y)$ false and $\text{son}(X,Y)$ false is $11 * 51 * 52 / 57^2 = 8.98$. These expected frequencies are indicated between brackets. Note that they sum to 57, but not to any of the other marginal frequencies (this would require more sophisticated models of independence, such as conditional independence).

As before, zeroes (i.e., countermodels) in the table correspond to clauses. Prime implicants are obtained by combining zeroes as much as possible, by projecting the table onto the appropriate 2 dimensions. We then obtain the following theory:

$\text{daughter}(X,Y); \text{son}(X,Y) : \neg\text{parent}(Y,X) . \quad (15.8\%)$
 $\text{parent}(Y,X) : \neg\text{daughter}(X,Y) . \quad (8.5\%)$
 $\text{parent}(Y,X) : \neg\text{son}(X,Y) . \quad (7.1\%)$
 $: \neg\text{daughter}(X,Y), \text{son}(X,Y) . \quad (0.9\%)$

Between brackets the expected relative frequency of counter-instances is indicated, which can be taken as a measure of the *novelty* of the clause with respect

to the marginal distributions. For instance, the fourth clause has low novelty because there are relatively few substitutions making `son(X,Y)` true, and the same holds for `daughter(X,Y)`. That no substitutions making both literals true can be found in the data may thus well be due to chance. By the same reasoning, the first clause gets high novelty, since from the marginal frequencies one would expect it to be quite easy to make both literals false.

This analysis interprets clauses in a classical way, since the confirmation of a clause is independent of its syntactical form. If we take a logic programming perspective the approach can be simplified to 2-dimensional tables that assess the dependence between body and head. We refer the interested reader to [44].

4 Knowledge Representation for ILP

Logic is a powerful and versatile knowledge representation formalism. However, its versatility also means that there are usually many different ways of representing the same knowledge. What is the best representation depends on the task at hand. In this section we discuss several ways of representing a particular predictive ILP task in logic, pointing out the strengths and weaknesses of each. As a running example we use a learning problem from [69]. The learning task is to discover low size-complexity Prolog programs for classifying trains as Eastbound or Westbound. The problem is illustrated in Figure 1.

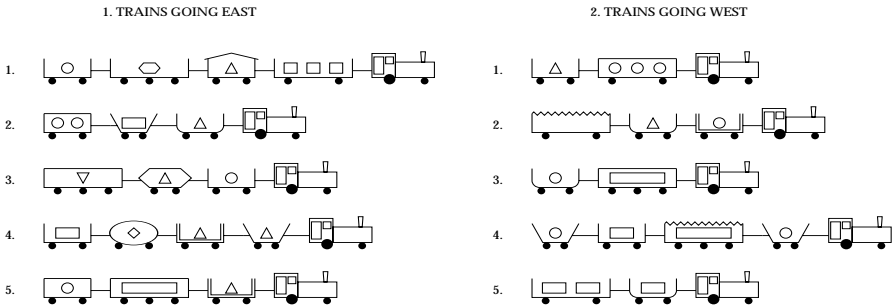


Fig. 1. The ten train East-West challenge.

Each train consists of 2-4 cars; the cars have attributes like shape (rectangular, oval, u-shaped, ...), length (long, short), number of wheels (2, 3), type of roof (none, peaked, jagged, ...), shape of load (circle, triangle, rectangle, ...), and number of loads (1-3). A possible rule distinguishing between eastbound and

westbound trains is ‘a train is eastbound if it contains a short closed car, and westbound otherwise’.

4.1 Datalog Representations

Datalog is a subset of Prolog in which the only functors are of arity 0 (i.e., constants). This simplifies inference as unification only needs to be performed between two variables, or between a variable and a constant. Similarly, it simplifies the specialisation and generalisation operators in ILP. The drawback is a loss of structure, as aggregation mechanisms such as lists are not available. Structured objects need to be represented indirectly, by introducing names for their parts.

A Datalog representation of the first train in Figure 1 is as follows.

```
eastbound(t1).

hasCar(t1,c11).      hasCar(t1,c12).
cshape(c11,rect).   cshape(c12,rect).
clength(c11,short). clength(c12,long).
croof(c11,none).    croof(c12,none).
cwheels(c11,2).     cwheels(c12,3).
hasLoad(c11,l11).   hasLoad(c12,l12).
lshape(l11,circ).    lshape(l12,hexa).
lnumber(l11,1).     lnumber(l12,1).

hasCar(t1,c13).      hasCar(t1,c14).
cshape(c13,rect).   cshape(c14,rect).
clength(c13,short). clength(c14,long).
croof(c13,peak).    croof(c14,none).
cwheels(c13,2).     cwheels(c14,2).
hasLoad(c13,l13).   hasLoad(c14,l14).
lshape(l13,tria).   lshape(l14,rect).
lnumber(l13,1).     lnumber(l14,3).
```

Using this representation, the above hypothesis would be written as

```
eastbound(T) :- hasCar(T,C), clength(C,short), not croof(C,none).
```

Testing whether this hypothesis correctly classifies the example amounts to proving the query `?-eastbound(t1)` from the hypothesis and the description of the example (i.e., all ground facts minus its classification).

Alternatively, we could represent an example by a ground clause:

```
eastbound(t1) :-
  hasCar(t1,c11), cshape(c11,rect), clength(c11,short),
  croof(c11,none), cwheels(c11,2),
  hasLoad(c11,l11), lshape(l11,circ), lnumber(l11,1),
```

```

hasCar(t1,c12),cshape(c12,rect),clength(c12,long),
    croof(c12,none),cwheels(c12,3),
    hasLoad(c12,l12),lshape(l12,hexa),lnumber(l12,1),
hasCar(t1,c13),cshape(c13,rect),clength(c13,short),
    croof(c13,peak),cwheels(c13,2),
    hasLoad(c13,l13),lshape(l13,tria),lnumber(l13,1),
hasCar(t1,c14),cshape(c14,rect),clength(c14,long),
    croof(c14,none),cwheels(c14,2),
    hasLoad(c14,l14),lshape(l14,rect),lnumber(l14,3).

```

From the logical point of view this representation is slightly odd because it doesn't actually assert the existence of train t_1 – only that, if t_1 existed and had the indicated properties, it would be eastbound. On the other hand, such hypothetical statements are all that is required for an induction algorithm, and we are not interested in rules referring to individual examples anyway. This representation also suggests an alternative way of testing whether a single-clause hypothesis covers an example, namely by a subsumption test.

Note that the body of each ground clause is a set of ground atoms, which can alternatively be seen as a Herbrand interpretation containing all facts describing a single example. Consequently, this setting is often referred to as *learning from interpretations* [19] (notice that this setting does not have to be restricted to Datalog, since the ground atoms in an interpretation may contain complex terms). The key point of this setting is that it allows us to keep all information pertaining to a single example together. In contrast, in the first Datalog representation facts belonging to different examples get mixed in a dataset. This is an important advantage of the ground clause or Herbrand interpretation representation, which increases the efficiency of mining algorithms significantly [5]. The term representations discussed in the next section are similarly individual-centred.

4.2 Term Representations

In full Prolog we can use terms to represent individuals. The following representations uses functors to represent cars and loads as tuples, and lists to represent a train as a sequence of cars.

```

eastbound([car(rect,short,none,2,load(circ,1)),
    car(rect,long,none,3,load(hexa,1)),
    car(rect,short,peak,2,load(tria,1)),
    car(rect,long,none,2,load(rect,3))]).

```

In this representation, the hypothesis given before is expressed as follows:

```
eastbound(T):-member(C,T),arg(2,C,short),not arg(3,C,none).
```

Here we use the built-in Prolog predicate $\text{arg}(N,T,A)$, which is true if A is the N -th argument of complex term T .

Strictly speaking, this representation is not equivalent to the previous ones because we now encode the order of cars in a train. We could encode the order of cars in the Datalog representation by using the predicates `hasFirstCar(T,C)` and `nextCar(C1,C2)` instead of `hasCar(T,C)`. Alternatively, we can ignore the order of cars in the term representation by only using the `member/2` predicate, effectively turning the list into a set. From the point of view of hypotheses, the two hypothesis representations are isomorphic: `hasCar(T,C)` corresponds to `member(C,T)`, `cLength(C,short)` corresponds to `arg(2,C,short)`, and `croof(C,none)` corresponds to `arg(3,C,none)`. Thus, Datalog and term representations look very different concerning examples, and very similar concerning hypotheses.

Like the ground Datalog clause representation, the term representation has the advantage that all information pertaining to an individual is kept together. Moreover, the structure of the terms can be used to guide hypothesis construction, as there is an immediate connection between the type of an individual and the predicate(s) used to refer to parts of the individuals. This connection between term structure and hypothesis construction is made explicit by using a strongly typed language [40]. The following representation uses a Haskell-like language called Escher, which is a higher-order logic and functional programming language [67].

```
eastbound :: Train->Bool;
type Train = [Car];
type Car = (CShape,CLength,CRoof,CWheels,Load);
data CShape = Rect | Hexa | ...;
data CLength = Long | Short;
data CRoof = None | Peak | ...;
type CWheels = Int;
type Load = (LShape,LNumber);
data LShape = Circ | Hexa | ...;
type LNumber = Int;

eastbound([(Rect,Short,None,2,(Circ,1)),
           (Rect,Long, None,3,(Hexa,1)),
           (Rect,Short,Peak,2,(Tria,1)),
           (Rect,Long, None,2,(Rect,3))]) = True;
```

The important part here is the type signature. The first line defines `eastbound` as a function mapping trains to booleans. The lines starting with `type` define type synonyms (i.e., the type signature could be rewritten without them). The lines starting with `data` define algebraic datatypes; here, they are simply enumerated types. The actual representation of an example is very similar to the Prolog term representation, except that it is an equation rather than a fact. Notice that functions are more natural to express classification rules than predicates.

The hypothesis is now expressed as follows:

```
eastbound(t) = (exists \c -> member(c,t) &&
                proj2(c)==Short && proj3(c)!=None)
```

Here, the phrase `exists \c ->` stands for explicit existential quantification of variable `c`, and `proj2` and `proj3` project on the second and third component of a 5-tuple representing a car, respectively. Again, the hypothesis is structurally similar to the Prolog one. However, the main point about strongly typed representations is that the type signature is available to the learning algorithm to guide hypothesis construction.

The term perspective gives us a clear view on the relation between attribute-value learning and first- and higher-order learning. In attribute-value learning, examples are represented by tuples of constants. Hypotheses are built by referring to one of the components of the tuple by means of projection, followed by a boolean condition on that component (e.g., being equal to a constant).² First-order representations such as Prolog generalise this by allowing lists and other recursive types, as well as an arbitrary nesting of subtypes (e.g., an individual could be a tuple, one component of which could be a list of tuples). Higher-order representations generalise this further by allowing sets and multisets.³

4.3 Database Representations

A third representation formalism for ILP is relational databases. This representation is clearly related to the Datalog representation: in particular, both representations refer to individuals and their parts by means of unique identifiers. However, there is also a close link with the term representation, as each complex type corresponds to a database relation, and the nesting of types corresponds to (chains of) foreign keys in the database.

A relational database representation is given in Figure 2. The train attribute in the CAR relation is a foreign key to *trainID* in TRAIN, and the car attribute in the LOAD relation is a foreign key to *carID* in CAR. Notice that the first foreign key is one-to-many, and the second one is one-to-one. An SQL version of the hypothesis discussed earlier is

```
SELECT DISTINCT TRAIN.trainID FROM TRAIN, CAR WHERE
    TRAIN.trainID = CAR.train AND
    CAR.shape = 'rectangle' AND
    CAR.roof != 'none'
```

² In practice this projection is not explicitly used, as any condition on a component of the tuple can be equivalently written as a condition on the tuple. The resulting rule will then have the same variable in all literals. Such rules could be called *semi-propositional*, as the only role of the variable is to distinguish hypotheses from examples. This explains why attribute-value learning is often loosely called propositional learning.

³ A set is equivalent to a predicate; passing around sets as terms requires a higher-order logic.

TRAIN

<i>trainID</i>	eastbound
<i>t1</i>	true

CAR

<i>carID</i>	cshape	clength	croof	cwheels	train
<i>c11</i>	rect	short	none	2	t1
<i>c12</i>	rect	long	none	3	t1
<i>c13</i>	rect	short	peak	2	t1
<i>c14</i>	rect	long	none	2	t1

LOAD

<i>loadID</i>	lshape	lnumber	car
<i>l11</i>	circ	1	c11
<i>l12</i>	hexa	1	c12
<i>l13</i>	tria	1	c13
<i>l14</i>	rect	3	c14

Fig. 2. A relational database representation of the East-West challenge.

This query performs a join of the TRAIN and CAR tables over *trainID*, selecting only rectangular closed cars. To prevent trains that have more than one such car to be included several times, the DISTINCT construct is used.

While this database representation does not seem to add much to the first Datalog representation from Section 4.1, it focuses attention on the fact that we really need a data model to describe the inherent structure of our data. Such a data model could be expressed, e.g., as an entity-relationship diagram (Figure 3), and plays the same role as the type signature in the strongly typed term representation. An alternative would be to use description logics to model the structure of the domain. Some work on learning description logic expressions is reported in [13], but note that this requires a different learning setting as description logic expressions can express concepts (intensional descriptions of classes of individuals) but not single individuals.

Like the first Datalog representation, the database representation has the disadvantage that examples are not easily separable. The term representations and the ground Datalog clause representations are superior in this respect. The term representation works very nicely on tree-structured data, but when the individuals are graphs (e.g., molecules) naming cannot be avoided in this representation either. Moreover, the term representation can be inflexible if we want to learn on a different level of individuals, e.g., if we want to learn on the level of cars rather than trains (the same holds for the ground Datalog clauses).

On the other hand, the strongly typed term representation provides a strong language bias, as hypothesis construction is guided by the structure of the individual. In particular, the strongly typed perspective advocates a distinction between two types of predicates:

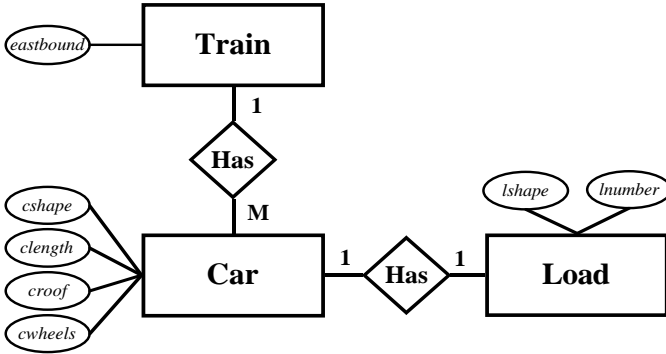


Fig. 3. Entity-relationship diagram for the East-West challenge.

1. *structural* predicates, which introduce variables, and
2. *utility* predicates (also called *properties*), which consume variables.

From the perspective of the entity-relationship data model, utility predicates correspond to attributes of entities, while structural predicates correspond to relationships between entities. This provides a useful language bias even in the Datalog representation; it has been used in [42] to upgrade the naive Bayes classifier to first-order logic.

4.4 Other Approaches to Language Bias

Below, we briefly review other approaches to fight the inherent complexity of ILP by imposing constraints, mostly syntactic in nature, on candidate hypotheses. Such constraints are grouped under the heading of *language bias* (there are other forms of biases that influence hypothesis selection; see [81] for an overview of declarative bias in ILP).

Essentially, the main source of complexity in ILP derives from the local variables in hypothesis clauses. In top-down systems, the branching factor of the specialisation operator increases with the number of variables in the clause. Typing is useful here, since it rules out many potential substitutions and unifications. Furthermore, one can simply put a bound on the number of distinct variables that can occur in a clause. In bottom-up systems, at some point one has to construct θ -LGG's for two or more ground clauses, which introduces many literals with variables occurring in the body but not in the head of the clause (existential variables). The approach of Golem is to restrict the introduction of existential variables by means of *ij*-determinacy, which enforces that every existential variable is uniquely determined by the preceding variables (*i* and *j* are depth parameters) [77].

Mode declarations are a well-known device from logic programming to describe possible input-output behaviour of a predicate definition. For instance, a sorting program will have a mode declaration of `sort(+list,-list)`, meaning that the first argument must be instantiated to a list. Progol uses extended mode declarations such as the following:

```
modeh(*,factorial(+int,-int)).
modeb(*,factorial(+int,-int)).
modeb(*,decr(+int,-int)).
modeb(*,mult(+int,+int,-int)).
```

A `modeh` declaration concerns a predicate that can occur in the head of a hypothesis clause, while `modeb` declarations relate to body literals. A set of mode declarations defines a mode language as follows: the head of the clause contains a predicate from a `modeh` declaration with arguments replaced by variables, and every body literal contains a predicate from a `modeb` declaration with arguments replaced by variables, such that every variable with mode `+type` is also of mode `+type` in the head, or of mode `-type` in a preceding literal. The mode language corresponding to the above mode declarations thus includes the clause

```
factorial(A,B):-decr(A,C),factorial(C,D),mult(A,D,B).
```

The asterisk `*` in the above mode declarations indicates that the corresponding literal can have any number of solutions; it may be bounded by a given integer. In addition one can apply a depth bound to a variable; e.g., in the clause just given the variable `D` has depth 2.

Refinement operators can be used as a language bias, since they can be restricted to generate only a subset of the language. For instance, a refinement operator can easily be modified to generate only singly-recursive or tail-recursive clauses. DLAB (declarative language bias) is a powerful language for specifying language bias [21]. Finally, we mention the use of clause schemata as a language bias. These are second-order clauses with predicate variables:

```
Q(X,Y):-P(X,Y).
Q(X,Y):-P(X,Z),Q(Z,Y).
```

Such schemata are used to constrain possible definitions of predicates; in this case it stipulates that any predicate that instantiates `Q` must be defined as the transitive closure of some other predicate.

5 State-of-the-Art ILP Techniques for Relational Data Mining

We continue to give a brief overview of state-of-the-art ILP techniques for relational data mining. Most of the outlined techniques are described in detail in [32]. This overview is limited to predictive and descriptive ILP techniques resulting in symbolic knowledge representations, excluding non-symbolic first-order

approaches such as relational instance-based learning [34], first-order reinforcement learning [31], and first-order Bayesian classifiers [42]. It has been suggested [27] to integrate the two main settings of predictive and descriptive ILP; in this integrated framework the learned theory is a combination of (predictive) rules and (descriptive) integrity constraints that restrict the consequences of these rules.

5.1 Predictive ILP

Learning of classification rules. This is the standard ILP setting that has been used in numerous successful predictive knowledge discovery applications. Well-known systems for classification rule induction include Foil [86], Golem [77] and Progol [80]. Foil is efficient and best understood, while Golem and Progol are less efficient but have been used in many of the successful ILP applications. Foil is a top-down learner, Golem is a bottom-up learner, and Progol uses a combined search strategy. All are mainly concerned with single predicate learning from positive and negative examples and background knowledge; in addition, Progol can also be used to learn from positive examples only. They use different acceptance criteria: compression, coverage/accuracy and minimal description length, respectively. The system LINUS [62,63], developed from a learning component of QuMAS [74], introduced the propositionalisation paradigm by transforming an ILP problem into a propositional learning task.

Induction of logical decision trees. The system Tilde [4] is a top-down decision tree induction algorithm. It can be viewed as a first-order upgrade of Quinlan's C4.5, employing logical queries in tree nodes which involves appropriate handling of variables. The main advantage of Tilde is its efficiency and capability of dealing with large numbers of training examples, which Tilde inherits from its propositional ancestors. Bowers *et al.* describe a decision tree learner that employs higher-order logic [7]. An important difference with Tilde is that Tilde constructs trees with single literals in the nodes, and thus local variables are shared among different nodes. In contrast, the higher-order learner constructs more complex features for each node, such that all variables are local to a node.

First-order regression. A regression task concerns prediction of a real-valued variable rather than a class. The relational regression task can be defined as follows: Given training examples as positive ground facts for the target predicate $r(Y, X_1, \dots, X_n)$, where the variable Y has real values, and background knowledge defining additional predicates, find a definition for $r(Y, X_1, \dots, X_n)$, such that each clause has a literal binding Y (assuming that X_1, \dots, X_n are bound). Typical background knowledge predicates include less-or-equal tests, addition, subtraction and multiplication. An approach to relational regression is implemented in the system FORS (First Order Regression System) [53] which performs top-down search of a refinement graph. In each clause, FORS can predict a value for the target variable Y as the output value of a background knowledge literal, as a constant, or as a linear combination of variables appearing in the clause (using linear regression).

Inductive Constraint Logic Programming. It is well known that Constraint Logic Programming (CLP) can successfully deal with numerical constraints. The idea of Inductive Constraint Logic Programming [89] is to benefit from the number-handling capabilities of CLP, and to use the constraint solver of CLP to do part of the search involved in inductive learning. To this end a maximally discriminant generalisation problem in ILP is transformed to an equivalent constraint satisfaction problem (CSP). The solutions of the original ILP problem can be constructed from the solutions of CSP, which can be obtained by running a constraint solver on CSP.

5.2 Descriptive ILP

Learning of clausal theories and association rules. In discovering full clausal theories, as done in the system Claudien [21], each example is a Herbrand model, and the system searches for the most general clauses that are true in all the models. Clauses are discovered independently from each other, which is a substantial advantage for data mining, as compared to the learning of classification rules (particularly learning of mutually dependent predicates in multiple predicate learning). In Claudien, search of clauses is limited by the language bias. Its acceptance criterion can be modified by setting two parameters: the requested minimal accuracy and minimal number of examples covered. In another clausal discovery system, Tertius [44], the best-first search for clauses is guided by heuristics measuring the “confirmation” of clauses. The Claudien system was further extended to Warmr [16,17] that enables learning of association rules from multiple relations.

First-order clustering. Top-down induction of decision trees can be viewed as a clustering method since nodes in the tree correspond to sets of examples with similar properties, thus forming concept hierarchies. This view was adopted in C0.5 [20], an upgrade of the Tilde logical decision tree learner. A relational distance-based clustering method is presented also in [58]. An early approach combining learning and conceptual clustering techniques was implemented in the system Cola [33]. Given a small (sparse) set of classified training instances and a set of unclassified instances, Cola uses Bisson’s conceptual clustering algorithm KBG on the entire set of instances, climbs the hierarchy tree and uses the classified instances to identify (single or disjunctive) class descriptions.

Database restructuring. The system Fender [92] searches for common parts of rules describing a concept, thus forming subconcept definitions to be used in the reformulation of original rules. The result is a knowledge base with new intermediate concepts and deeper inferential structure than the initial “flat” rulebase. The system Index [35] is concerned with the problem of determining which attribute dependencies (functional or multivalued) hold in the given relational database. The induced attribute dependencies can be used to obtain a more structured database. Both approaches can be viewed as doing predicate invention, where (user selected) invented predicates are used for theory restructuring. Various algorithms for discovery of database dependencies can be found in [41,88].

Subgroup discovery. The subgroup discovery task is defined as follows: given a population of individuals and a target property of those individuals we are interested in, find sufficiently large subgroups of the population that have a significantly different distribution with respect to the target property. The system Midos [98] guides the top-down search of potentially interesting subgroups using numerous user-defined parameters. The Tertius system [44] can also perform subgroup discovery. The Warmr system [16,17] can be used to find frequent queries, i.e., conjunctions of literals that have sufficiently many answers.

Learning models of dynamic systems. The automated construction of models of dynamic system may be aimed at qualitative model discovery. A recent qualitative model discovery system [48], using a Qsim-like representation, is based on Coiera's Genmodel to which signal processing capabilities have been added. The system LAGRANGE [29] discovers a set of differential equations from an example behaviour of a dynamic system. Example behaviours are specified by lists of measurements of a set of system variables, and background knowledge predicates enable the introduction of new variables as time derivatives, sines or cosines of system variables. New variables can be further introduced by multiplication.

6 Future Challenges for ILP

This section first presents some application challenges for ILP and continues with the technological advances that will be needed to deal with these challenges. We distinguish between short-term research challenges for ILP and longer-term challenges for ILP and machine learning in general. Finally, we address the connections with the areas of computational logic that may prove to be fruitful in future ILP research.

In our view, the most challenging application areas are in molecular biology, agents, personalised software applications, skill acquisition, natural language processing, information retrieval and text mining, analysis of music and multimedia data, as well as relational knowledge discovery applications in finance, e-commerce, banking, medicine, ecology, and others. For an overview of the state-of-the-art applications of ILP see [32], where also some future application challenges for ILP are indicated.

At present, molecular biology applications of ILP have come closest to practical relevance. Among the early applications was protein secondary structure prediction [78], followed by predicting drug activity through modelling structure-activity relations [77,55] and predicting the mutagenicity of aromatic and heteroaromatic nitro-compounds [94]. In these problems, which are of immediate practical interest, accuracies that are at least as good as the best previously known results have been obtained, as well as understandable and relevant new knowledge. Recent ILP applications in the area of molecular biology include prediction of rodent carcinogenicity bioassays, modelling structure-activity relations for modulating transmembrane calcium movement, pharmacophore discovery for ACE inhibition and diterpene structure elucidation. In the future there is consid-

erable potential for ILP applications using data produced by the human genome project, where the first successful ILP results have already been achieved [56,57].

6.1 Short-Term Research Challenges

ILP as a methodology for first-order learning. ILP has already developed numerous useful techniques for relational knowledge discovery. A recent research trend in ILP is to develop algorithms upgrading well-understood propositional machine learning techniques to first-order representations. Already developed techniques upgrading propositional learning algorithms include first-order decision tree learning [4,7], first-order clustering [20,58], relational genetic algorithms [46,66], first-order instance-based learning [34], first-order reinforcement learning [31] and first-order Bayesian classification [42]. It is expected that the adaptation of propositional machine learning algorithms to the first-order framework will continue also in the areas for which first-order implementations still do not exist. This should provide a full scale methodology for relational data mining based on future ILP implementations of first-order Bayesian networks, first-order neural networks, and other ILP upgrades of propositional machine learning techniques.

Improved robustness, efficiency and scaling-up of ILP algorithms.

This involves the development of learning algorithms that are robust with respect to noise, missing information etc., the development of standards for data and knowledge representation, standards for parameter settings, on-line transformers between different data formats, improved efficiency of learners, and the capacity of dealing with large datasets. Improved efficiency and scaling-up of ILP algorithms has to some extent already been achieved e.g., by the system Tilde [4] for induction of logical decision trees. Efficiency may be, on the one hand, achieved by effective coupling of ILP algorithms with database management systems, and on the other hand, by speeding-up the search of the lattice of clauses and speeding up of the testing of clause coverage involving repeated searches for proofs [6]. Speed-ups can also be achieved by employing sampling, stochastic search and stochastic matching procedures that are expected to be further developed in the future. Further speed-ups may be achieved by parallel processing, based on distributing the hypothesis space and testing competing hypotheses against the data independently and in parallel.

Multi-strategy learning and integration. The present data mining applications typically require data analysis to be performed by different machine learning algorithms, aimed at achieving best learning results. Multistrategy learning has shown that best results can be achieved by a combination of learning algorithms or by combining the results of multiple learners. Current simple and popular approaches involve bagging and boosting that employ redundancy to achieve better classification accuracy [9,45,87]. More sophisticated approaches will require the integration of different learners into knowledge discovery tools, standard statistical tools and spreadsheet packages and into software packages routinely used in particular applications. Integrated machine learning will have to be based also on a better understanding of the different types of problem domains and characteristics of learning algorithms best suited for the given data

characteristics. Mixing of different rules by the use of logic programming techniques also allows for combining multi-strategy and multi-source learning in a declarative way. Some of the existing techniques are inspired on contradiction removal methods originated in logic programming, others rely on recent work on updating logic programs with each other [2]. Logic program combination techniques may become more important in the near future.

Hierarchically structured learning and predicate invention. Learning from ‘flat’ datasets nowadays typically results in ‘flat’ hypotheses that involve no intermediate structure and no constructive induction/predicate invention. Despite substantial research efforts in this area challenging results can still be expected.

Criteria for the evaluation of hypotheses. Except for the standard measure of predictive accuracy, other evaluation measures need to be developed, e.g., ROC-based measures [85] and measures involving misclassification costs. Development of new measures is of particular importance for descriptive ILP systems that often lack such measures for the evaluation of results. Measures of similarity, distance measures, interestingness, precision, measures for outlier detection, irrelevance, and other heuristic criteria need to be studied and incorporated into ILP algorithms.

Criteria for the relevance of background knowledge. Background knowledge and previously learned predicate definitions should be stored for further learning in selected problem areas. One should be aware, however, that an increased volume of background knowledge may also have undesirable properties: not only that learning will become less efficient because of the increased hypothesis space, but given irrelevant information the results of learning may be less accurate. Therefore it is crucial to formulate criteria for evaluating the relevance of background knowledge predicates before they are allowed to become part of a library of background knowledge predicates for a given application area.

Learning from temporal data. ILP is to some extent able to deal with temporal information. However, specialised constructs should be developed for applications in which the analysis of a current stream of time labelled data represents an input to ILP. Experience from the area of temporal data abstraction could be used to construct higher-level predicates summarising temporal phenomena.

6.2 Long-Term Research Challenges

Some of the issues discussed in this section are relevant to ILP only, whereas others are relevant to machine learning in general. Some of these issues have been identified previously by Tom Mitchell in an article published in the Fall 1997 issue of the AI Magazine [71].

Analysis of comprehensibility. It is often claimed that for many applications comprehensibility is the main factor if the results of learning are to be accepted by the experts. Despite these claims and some initial investigations of intelligibility criteria for symbolic machine learning (such as Occam’s razor and

minimal description length criteria) there are few research results concerning the intelligibility evaluation by humans.

Building specialised learners and data libraries. Particular problem areas have particular characteristics and requirements, and not all learning algorithms are capable of dealing with these. This is a reason for starting to build specialised learners for different types of applications. This may involve also the development of special purpose reasoning mechanisms. In addition, libraries of ‘cleaned’ data, background knowledge and previously learned predicate definitions should be stored for further learning in selected problem areas. Notice that such libraries are currently being established for certain problem areas in molecular biology. This approach will lead to the re-usability of components and to extended example sets; these can also be obtained by systematic query answering and experimentation as part of ‘continuous’ learning, discussed next.

Continuous learning from ‘global’ datasets. Under this heading we understand the requirement for learning from various data sources, where data sources can be of various types, including propositional and relational tables, textual data, and hypermedia data including speech, images and video, including human expert interaction. This involves the issue of globality, i.e., learning from local datasets as well as referential datasets collected and maintained by the world’s best experts in the area, referential case bases of ‘outlier’ data as well as data that is publicly available on the web. Achieving the requirement of continuous and global learning will require also learning agents for permanent learning by theory revision from updated world-wide data, as well as the development of query agents that will be able to access additional information from the internet via query answering (invoked either by experts or by automatically extracting answers from WWW resources, possibly by invoking learning and active experimentation). Query agents may involve dynamic abductive querying on the web.

6.3 Specific Short-Term Challenges Related to Computational Logic

Constraint logic programming. As shown in the overview of techniques for predictive ILP in Section 5.1, the connection between ILP and CLP has already been established through the work on Inductive Constraint Logic Programming. ILP has recognised the potential of CLP number-handling and of the CLP constraint solving to do part of the search involved in inductive learning. Early work in this area by Page and Frisch, Mizoguchi and Ohwada in 1992, and more recent work by Sebag and Rouveirol [89] show the potential of merging ILP and CLP that has to be explored to a larger extent in the future. Due to the industrial relevance of these two areas of computational logic it is expected that the developments at their intersection may result in products of great industrial benefit.

Abduction. Other initiatives spanning different areas of computational logic have also identified the potential for mutual benefits. A series of workshops has

been organised on the relation and integration of abduction and induction, resulting in the first edited volume on the topic [43]. Early research in this direction by De Raedt (the system CLINT) and more recent work by Dimopoulos and Kakas [26] show the potential for merging these technologies. A new ILP framework and system, called ACL [52] for abductive concept learning has been developed and used to study the problems of learning from incomplete background data and of multiple predicate learning. More work in this area is expected in the future.

Higher-order logic. Some work towards the use of higher-order reasoning and the use of functional languages has also started, in particular using the declarative higher-order programming language Escher [67] for learning and hypothesis representation [40,66,7]. This work may be a first step towards a larger research initiative in using higher-order features in ILP.

Deductive databases. A tighter connection with deductive database technology has been advocated by De Raedt [22,24] introducing an inductive database mining query language that integrates concepts from ILP, CLP, deductive databases and meta-programming into a flexible environment for relational knowledge discovery in databases. Since the primitives of the language can easily be combined with Prolog, complex systems and behaviour can be specified declaratively. This type of integration of concepts from different areas of computational logic can prove extremely beneficial for ILP in the future. It can lead to a novel ILP paradigm of inductive logic programming query languages whose usefulness may be proved to be similar to those of constraint logic programming.

Other logic programming-based advances. Much work on logic program semantics in the past twelve years, culminating in the definition of well-founded semantics and stable model semantics, and subsequent elaborations could be considered in future ILP research, since they allow dealing with non-stratified programs, and 3-valuedness. Considerable work on knowledge representation and non-monotonic reasoning has been developed using such semantical basis. Also, recent work on constructive negation would allow inducing rules without fear of floundering, and generating exceptions to default negations which could then be generalised. Examples include learning together the positive and negative part of a concept where the learned theory is an extended logic program with classical negation [25,61,60] where a potential inconsistency in such a theory is resolved by learning priorities amongst contradictory rules. Argumentation semantics and procedures are also likely to be useful for composing rules learned separately from several sources, algorithms, or strategies.

The work in logic programming on preferences [10,11] is bound to be of interest when combining rules, and even more so because user preferences might be learned from instances of user choice and rejection. This may turn out to be crucial for information gathering on the basis of user preferences. Fuzzy logic programming may become important in the future for fuzzifying such induced preference rules, as well as generalised annotated programs [54] which allow for different degrees of contradiction to be expressed.

Moreover, the implementational techniques of tabling in logic programming have matured and prove quite useful [99]. In ILP they may save considerable recomputation because results are memoized in an efficient way. Indeed, in ILP each time a clause is abstracted or refined it has to be tested again with the evidence, though many literals in the clause, and surely the background, are the same, so that part of the computation is repeated. This is even more important when learned programs become deeper, i.e., not shallow.

7 Concluding Remarks

Research areas that have strongly influenced ILP research are (apart from computational logic): machine learning, data mining and knowledge discovery in databases, and computational learning theory.

ILP has its roots in machine learning, and most of ILP researchers have done machine learning research before entering ILP. Machine learning has always provided the basic research philosophy where experimental evaluation and applications play a key role in the development of novel techniques and tools. This was the case in the early days of ILP and remains so today. Important influences from data mining and knowledge discovery in databases concern mainly the development of new ILP algorithms in the descriptive ILP setting, as well as the emphasis on scaling-up ILP algorithms to deal with large relational databases. Computational learning theory has helped ILP to better understand the learnability issues and provided some basic learning algorithms that were studied and adapted for the needs of ILP.

From the perspective of this chapter it is interesting to analyse the impact of computational logic and logic programming on ILP developments. Both played an extremely important role in early ILP research. Besides providing a framework that helped to develop the theory of ILP, it provided the well-studied representational formalisms and an initially challenging application area of program synthesis. Due to the difficulty of this application task that can not be solved without very strong biases and restrictions on the hypothesis language, program synthesis has become substantially less popular in recent years.

The analysis of theoretical papers in the proceedings of ILP workshops in 1991–1998 by De Raedt [23] indicates that about one third of accepted papers are related to logic programming. The main issues studied in these papers are inference rules, program synthesis, negation, constraint logic programming, abduction, and implementation. One important observation made by De Raedt is that the theory of ILP does not follow recent developments in logic programming and computational logic but, to a large extent, uses the well-established results obtained in early logic programming research. On the other hand, advanced logic programming techniques may become increasingly important once ILP starts seriously addressing difficult learning problems in natural language processing, where recursion, negation, higher-order logic, and other issues requesting a strong theoretical foundation in logic programming will come into play again.

Acknowledgements

We thank two anonymous reviewers for their helpful comments. Part of the material in Sections 2–4 is based on a tutorial given by the first author at the First International Conference on Computational Logic (CL-2000). The section on the state-of-the-art ILP techniques reflects some of the research results achieved in the ESPRIT projects no. 6020 ILP (1992–95) and no. 20237 ILP2 (1996–99). The section on future challenges for ILP has been to a large extent influenced by the panel discussion with panelists Luc De Raedt, Stephen Muggleton and Nada Lavrač, organised by John Lloyd as part of the workshop “Logic Programming and Machine Learning: A Two-way Connection”, organised at JICSLP’98 in Manchester in June 1998 under auspices of the CompulogNet area “Computational Logic and Machine Learning”. The interested reader can find further directions for ILP research in David Page’s invited talk at CL-2000 [82]. This work has been supported by the Network of Excellence in Inductive Logic Programming *ILPnet2*, the EU-funded project Data Mining and Decision Support for Business Competitiveness: A European Virtual Enterprise (IST-1999-11495), the British Royal Society, the British Council (Partnership in Science PSP 18), and the Slovenian Ministry of Science and Technology.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetski-Shapiro, P. Smyth, and R. Uthurusamy (eds.), *Advances in Knowledge Discovery and Data Mining*, pp. 307–328. AAAI Press, 1996.
2. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic logic programming, In A. Cohn, L. Schubert and S. Shapiro (eds.), *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 98–109. Morgan Kaufmann, 1998.
3. D. Angluin, M. Frazier, and L. Pitt. Learning conjunctions of Horn clauses. *Machine Learning*, 9(2/3): 147–164, 1992.
4. H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence* 101(1-2): 285–297, June 1998.
5. H. Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3(1): 59–93, 1999.
6. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In J. Cussens and A. Frisch (eds.), *Proceedings of the Tenth International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence 1866, pp. 60–77. Springer-Verlag, 2000.
7. A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. Classification of individuals with complex structure. In P. Langley (ed.), *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 81–88. Morgan Kaufmann, 2000.
8. I. Bratko and S. Muggleton. Applications of Inductive Logic Programming. *Communications of the ACM* 38(11): 65–70, November 1995.
9. L. Breiman. Bagging predictors. *Machine Learning* 24(2): 123–140, 1996.

10. G. Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *Journal of Artificial Intelligence Research*, 4: 19–36, 1996.
11. G. Brewka and T. Eiter. Preferred answer sets. In A. Cohn, L. Schubert and S. Shapiro (eds.), *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 89–97. Morgan Kaufmann, 1998.
12. W.W. Cohen. Recovering software specifications with inductive logic programming. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 142–148. The MIT Press, 1994.
13. W.W. Cohen and H. Hirsh. Learning the CLASSIC Description Logic: Theoretical and Experimental Results. In J. Doyle, E. Sandewall, and P. Torasso (eds.), *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 121–133. Morgan Kaufmann, 1994.
14. J. Cussens. Notes on inductive logic programming methods in natural language processing (European work). Unpublished manuscript, 1998. ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/ilp98tut.ps.gz.
15. J. Cussens and S. Džeroski (eds.). *Learning Language in Logic*. Lecture Notes in Artificial Intelligence 1925, Springer-Verlag, 2000.
16. L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In S. Džeroski and N. Lavrač (eds.), *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence 1297, pp. 125–132. Springer-Verlag, 1997.
17. L. Dehaspe, H. Toivonen, and R.D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Pietetsky-Shapiro (eds.), *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, pp. 30–36. AAAI Press, 1998.
18. L. De Raedt (ed.). *Advances in Inductive Logic Programming*. IOS Press, 1996.
19. L. De Raedt. Logical settings for concept-learning. *Artificial Intelligence*, 95(1): 187–201, 1997.
20. L. De Raedt and H. Blockeel. Using logical decision trees for clustering. In N. Lavrač and S. Džeroski (eds.), *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence 1297, pp. 133–140. Springer-Verlag, 1997.
21. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26(2/3): 99–146, 1997.
22. L. De Raedt. An inductive logic programming query language for database mining (extended abstract). In J. Calmet and J. Plaza (eds.), *Proceedings of the Fourth Workshop on Artificial Intelligence and Symbolic Computation*, Lecture Notes in Artificial Intelligence 1476. Springer-Verlag, 1998.
23. L. De Raedt. A perspective on inductive logic programming. In K. Apt, V. Marek, M. Truszczyński, and D.S. Warren (eds.), *The logic programming paradigm: current trends and future directions*. Springer-Verlag, 1999.
24. L. De Raedt. A logical database mining query language. In J. Cussens and A. Frisch, *Proceedings of the Tenth International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence 1866, pp. 78–92. Springer-Verlag, 2000.
25. Y. Dimopoulos and A.C. Kakas. Learning non-monotonic logic programs: learning exceptions. In N. Lavrač and S. Wrobel (eds.), *Proceedings of the Eighth European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence 912, pp. 122–138. Springer-Verlag, 1995.
26. Y. Dimopoulos and A.C. Kakas. Abduction and inductive learning. In [18], pp. 144–171.

27. Y. Dimopoulos, S. Džeroski, and A.C. Kakas. Integrating Explanatory and Descriptive Induction in ILP. In M.E. Pollack (ed.), *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 900–907. Morgan Kaufmann, 1997.
28. B. Dolšak and S. Muggleton. The application of inductive logic programming to finite-element mesh design. In [76], pp. 453–472.
29. S. Džeroski and L. Todorovski. Discovering dynamics: From inductive logic programming to machine discovery. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 97–103. Morgan Kaufmann, 1993.
30. S. Džeroski and I. Bratko. Applications of Inductive Logic Programming. In [18], pp. 65–81.
31. S. Džeroski, L. De Raedt, and H. Blockeel. Relational reinforcement learning. In J. Shavlik (ed.), *Proceedings of the Fifteenth International Conference on Machine Learning*, pp. 136–143. Morgan Kaufmann, 1998.
32. S. Džeroski and N. Lavrač, eds. *Relational Data Mining*. Springer-Verlag, 2001. In press.
33. W. Emde. Learning of characteristic concept descriptions from small sets to classified examples. In F. Bergadano and L. De Raedt (eds.), *Proceedings of the Seventh European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence 784, pp. 103–121. Springer-Verlag, 1994.
34. W. Emde and D. Wettschereck. Relational instance-based learning. In L. Saitta (ed.), *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 122–130. Morgan Kaufmann, 1996.
35. P.A. Flach. Predicate invention in inductive data engineering. In P. Brazdil (ed.), *Proceedings of the Sixth European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence 667, pp. 83–94. Springer-Verlag, 1993.
36. P.A. Flach. *Simply Logical – intelligent reasoning by example*. John Wiley, 1994.
37. P.A. Flach. *Conjectures – an inquiry concerning the logic of induction*. PhD thesis, Tilburg University, April 1995.
38. P.A. Flach. Rationality postulates for induction. In Y. Shoham (ed.), *Proceedings of the Sixth International Conference on Theoretical Aspects of Rationality and Knowledge*, pp. 267–281. Morgan Kaufmann, 1996.
39. P.A. Flach. Normal forms for Inductive Logic Programming. In N. Lavrač and S. Džeroski (eds.), *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence 1297, pp. 149–156. Springer-Verlag, 1997.
40. P.A. Flach, C. Giraud-Carrier, and J.W. Lloyd. Strongly typed inductive concept learning. In D. Page (ed.), *Proceedings of the Eighth International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence 1446, pp. 185–194. Springer-Verlag, 1998.
41. P.A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3): 139–160, November 1999.
42. P.A. Flach and N. Lachiche. 1BC: A first-order Bayesian classifier. In S. Džeroski and P.A. Flach (eds.), *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence 1634, pp. 92–103. Springer-Verlag, 1999.
43. P.A. Flach and A.C. Kakas (eds.) *Abduction and Induction: Essays on their Relation and Integration*. Kluwer, 2000.
44. P.A. Flach and N. Lachiche. Confirmation-guided discovery of first-order rules with Tertius. *Machine Learning*, 42(1/2): 61–95, 2001.

45. Y. Freund and R.E. Shapire. Experiments with a new boosting algorithm. In L. Saitta (ed.), *Proceedings of the Thirteenth International Conference on Machine Learning*, 148–156. Morgan Kaufmann, 1996.
46. A. Giordana and C. Sale. Learning structured concepts using genetic algorithms. In D. Sleeman (ed.), *Proceedings of the Ninth International Workshop on Machine Learning*, pp. 169–178. Morgan Kaufmann, 1992.
47. G. Gottlob. Subsumption and implication. *Information Processing Letters* 24: 109–111, 1987.
48. D.T. Hau and E.W. Coiera. Learning qualitative models of dynamic systems. *Machine Learning*, 26(2/3): 177–212, 1997.
49. N. Helft. Induction as nonmonotonic inference. In R.J. Brachman, H.J. Levesque, and R. Reiter (eds.), *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pp. 149–156. Morgan Kaufmann, 1989.
50. P. Idestam-Almqvist. *Generalization of clauses*. PhD thesis, Stockholm University, October 1993.
51. P. Idestam-Almqvist. Generalization of clauses under implication. *Journal of Artificial Intelligence Research*, 3: 467–489, 1995.
52. A.C. Kakas and F. Riguzzi. Learning with abduction. In S. Džeroski and N. Lavrač (eds.), *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence 1297, pp. 181–188. Springer-Verlag, 1997.
53. A. Karalič and I. Bratko. First-order regression. *Machine Learning*, 26(2/3): 147–176, 1997.
54. M. Kifer and V.S. Subrahmanian. Generalized annotated logic programs. *Journal of Logic Programming*, 1992.
55. R.D. King, S. Muggleton, R. Lewis, and M.J.E. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. In *Proceedings of the National Academy of Sciences of the USA* 89(23): 11322–11326, 1992.
56. R.D. King, A. Karwath, A. Clare, and L. Dehaspe. Genome scale prediction of protein functional class from sequence using data mining. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, pp. 384–398. ACM Press, New York, 2000.
57. R.D. King, A. Karwath, A. Clare, and L. Dehaspe. Accurate prediction of protein functional class in the *M.tuberculosis* and *E.coli* genomes using data mining. *Yeast (Comparative and Functional Genomics)*, 17: 283–293, 2000.
58. M. Kirsten and S. Wrobel. Relational distance-based clustering. In D. Page (ed.) *Proceedings of the Eighth International Conference on Inductive Logic Programming*, pp. 261–270, Lecture Notes in Artificial Intelligence 1446. Springer-Verlag, 1998.
59. P. van der Laag. *An analysis of refinement operators in Inductive Logic Programming*. PhD Thesis, Erasmus University Rotterdam, December 1995.
60. E. Lamma, F. Riguzzi, and L. M. Pereira. Agents learning in a three-valued logical setting. In A. Panayiotopoulos (ed.), *Proceedings of the Workshop on Machine Learning and Intelligent Agents*, in conjunction with *Machine Learning and Applications, Advanced Course on Artificial Intelligence (ACAI-99)*, Chania, Greece, 1999.
61. E. Lamma, F. Riguzzi, and L. M. Pereira. Strategies in combined learning via Logic Programs. *Machine Learning*, 38(1/2): 63–87, 2000.

62. N. Lavrač, S. Džeroski, and M. Grobelnik. Learning nonrecursive definitions of relations with LINUS. In Y. Kodratoff (ed.) *Proceedings of the Fifth European Working Session on Learning*, Lecture Notes in Artificial Intelligence 482, pp. 265–281. Springer-Verlag, 1991.
63. N. Lavrač and S. Džeroski. *Inductive Logic Programming: techniques and applications*. Ellis Horwood, 1994.
64. N. Lavrač, S. Džeroski, and I. Bratko. Handling imperfect data in Inductive Logic Programming. In [18], pp. 48–64.
65. N. Lavrač and P.A. Flach. An extended transformation approach to Inductive Logic Programming. *ACM Transactions on Computational Logic*, 2(4): 458–494, 2001.
66. C.J. Kennedy. *Strongly typed evolutionary programming*. PhD Thesis, University of Bristol, 2000.
67. J.W. Lloyd. Programming in an integrated functional and logic programming language. *Journal of Functional and Logic Programming*, 1999(3).
68. D.W. Loveland and G. Nadathur. Proof procedures for logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5, D.M. Gabbay, C.J. Hogger, and J.A. Robinson (eds.), Oxford University Press, pp. 163–234, 1998.
69. D. Michie, S. Muggleton, D. Page, and A. Srinivasan. To the international computing community: A new East-West challenge. Technical report, Oxford University Computing laboratory, Oxford, UK, 1994.
70. T.M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
71. T.M. Mitchell. Does machine learning really work? *AI Magazine* 18 (3): 11–20, 1997.
72. F. Mizoguchi, H. Ohwada, M. Daidoji, and S. Shirato. Using inductive logic programming to learn classification rules that identify glaucomatous eyes. In N. Lavrač, E. Keravnou, and B. Zupan (eds.), *Intelligent Data Analysis in Medicine and Pharmacology*, pp. 227–242. Kluwer, 1997.
73. R.J. Mooney and M.E. Califf. Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research* 3: 1–24, 1995.
74. I. Mozetič. Learning of qualitative models. In I. Bratko and N. Lavrač (eds.) *Progress in Machine Learning*, pp. 201–217. Sigma Press, 1987.
75. S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(4): 295–317, 1991. Also in [76], pp. 3–27.
76. S. Muggleton (ed.). *Inductive Logic Programming*. Academic Press, 1992.
77. S. Muggleton and C. Feng. Efficient induction of logic programs. In [76], pp. 281–298.
78. S. Muggleton, R.D. King, and M.J.E. Sternberg. Protein secondary structure prediction using logic. *Protein Engineering* 7: 647–657, 1992.
79. S. Muggleton and L. De Raedt. Inductive Logic Programming: theory and methods. *Journal of Logic Programming*, 19/20: 629–679, 1994.
80. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13: 245–286, 1995.
81. C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in Inductive Logic Programming. In [18], pp. 82–103.
82. D. Page. ILP: Just do it. In J.W. Lloyd (ed.), *Proceedings of the First International Conference on Computational Logic*, Lecture Notes in Artificial Intelligence 1861, pp. 25–40. Springer-Verlag, 2000.
83. G. Plotkin. A note on inductive generalisation. *Machine Intelligence* 5, B. Meltzer and D. Michie (eds.), pp. 153–163. North-Holland, 1970.

84. G. Plotkin. A further note on inductive generalisation. *Machine Intelligence 6*, B. Meltzer and D. Michie (eds.), pp. 101–124. North-Holland, 1971.
85. F. Provost and T. Fawcett. Robust classification for imprecise environments. *Machine Learning* 42(3): 203–231, 2001.
86. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3): 239–266, 1990.
87. J.R. Quinlan. Boosting, bagging, and C4.5 . In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 725–730. AAAI Press, 1996.
88. I. Savnik and P.A. Flach. Discovery of multivalued dependencies from relations. *Intelligent Data Analysis*, 4(3,4): 195–211, 2000.
89. M. Sebag and C. Rouveirol. Constraint Inductive Logic Programming. In [18], pp. 277–294.
90. E.Y. Shapiro. *Inductive inference of theories from facts*. Technical Report 192, Computer Science Department, Yale University, 1981.
91. E.Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.
92. E. Sommer. Rulebase stratifications: an approach to theory restructuring. In S. Wrobel (ed.), *Proceedings of the Fourth International Workshop on Inductive Logic Programming*, GMD-Studien 237, pp. 377–390, 1994.
93. A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel (ed.), *Proceedings of the Fourth International Workshop on Inductive Logic Programming*, GMD-Studien 237, pp. 217–232, 1994.
94. A. Srinivasan, R.D. King, S. Muggleton, and M.J.E. Sternberg. Carcinogenesis prediction using inductive logic programming. In N. Lavrač, E. Keravnou, and B. Zupan (eds.), *Intelligent Data Analysis in Medicine and Pharmacology*, pp. 243–260. Kluwer, 1997.
95. I. Stahl. Compression measures in ILP. In [18], pp. 295–307.
96. L. Valiant. A theory of the learnable. *Communications of the ACM* 27: 1134–1142, 1984.
97. I.H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques with Java implementations*. Morgan Kaufman, 2000.
98. S. Wrobel. An algorithm for multi-relational discovery of subgroups. In *Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery*, pp. 78–87. Springer-Verlag, 1997.
99. XSB Group Home Page: <http://www.cs.sunysb.edu/~sbprolog/>.