The logic of learning: a brief introduction to Inductive Logic Programming

Peter A. Flach
Department of Computer Science, University of Bristol
Merchant Venturers Building, Woodland Road, Bristol BS8 1UB, UK
Peter.Flach@cs.bris.ac.uk, http://www.cs.bris.ac.uk/~flach/

August 5, 1998

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. The aim of concept learning is to discover, from a given set of pre-classified examples, a set of 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

\[
\text{IF Outlook = Sunny AND Humidity = High THEN PlayTennis = No}
\]

When objects are structured and consist of several related parts, we need variable co-referencing. In the 1980’s machine learning researchers started exploring the use of logic programming representations, which led to the establishment of inductive logic programming (ILP) as a subdiscipline of machine learning. 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 [2, 6] for an overview of recent applications.

This paper is intended to provide an introduction to ILP. We will both review some of the established approaches to Horn clause induction (Section 2), and recent work on induction of integrity constraints (Section 3).

2 Horn clause induction

The problem of Horn clause induction is to construct a definition of a predicate from a subset of its ground instances. Although it is not impossible to learn from positive examples only, negative

\[
\text{IF Outlook = Sunny AND Humidity = High THEN PlayTennis = No}
\]
Definition 1 (Horn clause induction) A Horn clause induction problem is defined as follows:

Given:
\(P\): ground facts to be entailed (positive examples);
\(N\): ground facts not to be entailed (negative examples);
\(B\): a set of predicate definitions (background theory);
\(\mathcal{L}\): the hypothesis language;

Find a predicate definition \(H \in \mathcal{L}\) (hypothesis) such that
1. for every \(p \in P\): \(B \cup H \models p\) (completeness)
2. for every \(n \in N\): \(B \cup H \not
\models n\) (consistency)

For instance, the task might be to learn a definition for the predicate \(r/3\) from the following examples (and empty background theory).

Positive examples\(^1\):
- \(r([2,1],[3],[1,2,3]).\)
- \(r([a],[],[a]).\)
- \(r([1],[2,3],[1,2,3]).\)
- \(r([],[a],[a]).\)
- \(r([],[],[]).\)

Negative examples:
- \(r([1,2],[3],[1,2,3]).\)
- \(r([a],[a],[a]).\)
- \(r([],[],[]).\)

Induction systems may either process all examples in one run or incrementally one-by-one.

Clearly the problem as stated here is underconstrained, since for instance the extensional definition consisting of the positive examples only satisfies completeness and consistency. Whereas in program synthesis one can usually check the induced definition by inspection, in classification tasks the main requirement is that the hypothesis performs well on new, previously unseen instances. Various heuristic methods for estimating the performance of a hypothesis on the whole population will pass the review later in this section.

Broadly speaking, there are two approaches to the problem of Horn clause induction. One can either start from short clauses, progressively adding literals to their bodies as long as they are found

\(^1\)The intended predicate is non-naive reverse with an accumulator as second argument.
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, I will
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:

```
event | action | clause
---|---|---
+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 system initialises with the most general definition. 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 predic-
te 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 [39, 40],
an ILP system avant la lettre (the term ‘inductive logic programming’ was coined in 1990 by
Muggleton [28]). 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 [21] for an extensive analysis of refinement opera-
tors). A much simplified Prolog implementation of MIS can be found in [7]. Another well-known
top-down system is Quinlan’s FOIL [36].
2.2 Generality

As the previous example shows, clauses can be specialised in two ways: by applying a substitution, and by adding a body literal. More formally, the underlying structure can be defined as follows.

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

θ-subsumption is reflexive and transitive, but not antisymmetric (e.g. $p(X) : -q(X)$ and $p(X) : -q(Y), q(Y)$ 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 greater lower bound under θ-subsumption. We will refer to the least upper bound of two clauses under θ-subsumption as their θLGG (least general generalisation). Note that the lattice does contain infinite descending chains.

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

\[
\begin{align*}
nat(s(X)) & : -nat(X). \\
nat(s(s(Y))) & : -nat(Y). \\
nat(s(s(Z))) & : -nat(s(Z)).
\end{align*}
\]

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 $s(Z)$ for $X$) but not the second. Gottlob characterises the distinction between θ-subsumption and entailment [16]: 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 defined. 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 [19, 20].

\(^2\)This definition, and the term θ-subsumption, was introduced in the context of induction by Plotkin [34, 35]. 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 [25].
2.3 Bottom-up induction

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

\[
\begin{align*}
  a([1,2],[3,4],[1,2,3,4]). & \quad a([2],[3,4],[2,3,4]). \\
  a([a],[], [a]) & \quad a([],[],[]).
\end{align*}
\]

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 \(a/3\):

\[
\begin{align*}
  a([1,2],[3,4],[1,2,3,4]):= & \quad a([2],[3,4],[2,3,4]). \\
  a([a],[], [a]):= & \quad a([],[],[]).
\end{align*}
\]

All that remains to be done is to construct the \(\theta\)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:

\[
\begin{align*}
  a([A|B],C,[A|D]):= & \quad a(B,C,D).
\end{align*}
\]

which is easily recognised as the recursive clause in the standard definition of \(\text{append}/3\).

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:

\[
\begin{align*}
  a([1,2],[3,4],[1,2,3,4]):= & \quad a([1,2],[3,4],[1,2,3,4]), a([a],[], [a]), a([],[],[]), a([2],[3,4],[2,3,4]). \\
  a([a],[], [a]):= & \quad a([1,2],[3,4],[1,2,3,4]), a([a],[], [a]), a([],[],[]), a([2],[3,4],[2,3,4]).
\end{align*}
\]

Since bodies of clauses are, logically speaking, unordered, the \(\theta\)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:
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 head literal from the body, 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

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

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 [27] (again, a much simplified Prolog implementation can be found in [7]). 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 [31]. 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 below) 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 section on heuristics below). Progol is thus a hybrid bottom-up/top-down system. It has been successfully applied to a number of scientific discovery problems.
2.4 Language bias

It should be stressed that Horn clause induction is a difficult problem. Practical ILP systems fight the inherent complexity of the problem by imposing all sorts of 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 [32] for an overview of declarative bias in ILP).

Essentially, the main source of complexity in ILP derives from the 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 \( \theta \)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) [27].

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 \( \text{sort}(+\text{list}, -\text{list}) \), meaning that the first argument must be instantiated to a list. Progol uses extended mode declarations such as the following:

\[
\begin{align*}
\text{modeh}(\ast, \text{fact}(+\text{int}, -\text{int})). \\
\text{modeb}(\ast, \text{fact}(+\text{int}, -\text{int})). \\
\text{modeb}(\ast, \text{decr}(+\text{int}, -\text{int})). \\
\text{modeb}(\ast, \text{mult}(+\text{int}, +\text{int}, -\text{int})).
\end{align*}
\]

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

\[
\text{fact}(A, B) : -\text{decr}(A, C), \text{fact}(C, D), \text{mult}(A, D, B).
\]

The asterisk \( \ast \) 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 \( B \) 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 [5]. Finally, I 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.

2.5 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. Furthermore, heuristics have the additional benefit of making an induction algorithm noise-tolerant. We can only scratch the surface of the topic here — for overviews see [22, Chapter 8] or [23].

Accuracy estimates. 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 significance of the heuristic used.

The Bayesian approach. 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 I consider a particular hypothesis to be very unlikely, but certain evidence to be very likely given that hypothesis. If subsequently I indeed observe that evidence, this will increase my 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) maybe technically simple but hard to justify.

**Compression-based heuristics.** Finally, there is the compression approach [41]. 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 a posteriori 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 Induction of integrity constraints

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 from consequences and non-consequences (Definition 1). However, this definition reflects the fact 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 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 loose 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.

3.1 Descriptive induction

While the Horn clause induction setting of Definition 1 is sometimes referred to as explanatory induction (since the induced theory provides in a sense an explanation for the classification of the examples), the problem of induction of integrity constraints is referred to as descriptive or confirmatory induction. Since different authors differ in their formalisation of descriptive induction, I will not attempt a general definition here but rather point out two global characteristics.

In general it can be said that, while explanatory 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 [42, 1].

Secondly, 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’ [17]. Sometimes this enables one to treat the data as specifying one preferred or minimal model, and develop the hypothesis from that starting point. Metalogical properties of this form of inductive reasoning are therefore similar to those of reasoning with rules that tolerate exceptions [10, 11].

3.2 Induction of attribute dependencies

Attribute dependencies such as functional dependencies are forms of intensional knowledge that can be successfully induced from extensional data. Even if these dependencies can typically be
formulated as Horn rules, they are clearly not intended for classification (unless one is prepared to see, say, functional dependencies as part of the definition of equality!). Induction of attribute dependencies thus calls for the alternative framework of descriptive induction. Here I will briefly introduce my own work on induction of functional and multivalued dependencies (see [9] or [10, Chapter 8] for details; [26, 18] give alternative algorithms for discovery of functional dependencies).

Typically it is sufficient to learn functional and multivalued dependencies from single relations. The tuples can be supplied incrementally one-by-one or as a complete relation. There is an interesting difference between functional and multivalued dependencies: functional dependencies are violated by a pair of tuples in the relation, while multivalued dependencies can only be violated by a pair of tuples in the relation and a third tuple known to be not in the relation. Consequently, negative tuples are essential for incremental induction of multivalued dependencies (in my approach they were obtained through queries). Clearly in the non-incremental approach negative tuples are obtained through the Closed World Assumption.

Attribute dependencies, being logical formulae, can be ordered according to generality (a simplified form of $\theta$-subsumption). As in the case of explanatory induction, one can therefore have top-down and bottom-up approaches. In the top-down approach [8] one starts with the set of most general dependencies, specialising them until they are no longer violated by the data. In this way one obtains a cover for the set of satisfied dependencies (although the cover may contain some redundant elements). An alternative bottom-up approach runs as follows [38]. From the data one first constructs a negative cover, which is the set of least general violated dependencies (this can be done in $O(n^2)$ steps in the case of functional dependencies, and $O(n^3)$ steps in the case of multivalued dependencies). The positive cover can then be constructed from the negative cover, without reference to the tuples.

### 3.3 Clausal discovery

CLAUDIEN (clausal discovery engine) is a system for discovery of full clausal theories [3, 5]. Each example is a Herbrand interpretation, and the system searches for most general clauses that are true in all interpretations. The following example is taken from [5].

**Examples:**

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

**Hypothesis:**

```
gorilla(X) :- female(X)
gorilla(X) :- male(X)
```
Table 1: A feature table.

<table>
<thead>
<tr>
<th></th>
<th>female(X)</th>
<th>male(X)</th>
<th>gorilla(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>richard, fred</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>liz, ginger</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

\[ 
\text{male}(X);\text{female}(X);:\text{gorilla}(X) \\
:;\text{male}(X),\text{female}(X) \\
\]

Given that the hypothesis language only contains predicates that are present in the examples, this is a correct solution.

One way to see that is by means of DNF to CNF conversion [12]. From the examples we construct a feature table, which is a sort of generalised truth-table (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. The desired clausal theory can now be found by constructing the prime implicants of the countermodels and negating them. This yields the following theory:

\[ 
\text{gorilla}(X). \\
\text{male}(X);\text{female}(X). \\
:;\text{male}(X),\text{female}(X). \\
\]

\text{CLAUDIEN} requires all clauses to be range-restricted, and therefore adds literals to the bodies of the first and second clauses. The same effect can be obtained by adding a non-female, non-male non-gorilla to the examples.

The task addressed by the \text{CLAUDIEN} system is a kind of unsupervised learning closely related to data mining. An important difference with the classification-oriented form of ILP is that here each clause can be discovered independently of the others. Not only does this allow a parallel implementation [5], it also 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.
3.4 Heuristics for descriptive induction

One problem experienced by many data mining algorithms is that they typically produce large amounts of rules (be they attribute dependencies, association rules, or clauses). A possible solution would be to apply best-first search such that rules which maximise a certain heuristic are output first. We are currently investigating an approach based on the theory of statistical independence. We illustrate the approach by an example.

Suppose we are considering the literals daughter(X,Y), son(X,Y), and parent(Y,X). As in Table 1 we count the number of substitutions for each possible truth-value assignment, but instead of a truth table 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 parent(Y,X) true, daughter(X,Y) false and son(X,Y) false is $11 \times 51 \times 52 / 57^2 = 8.98$. 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 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:

\[
\begin{align*}
\text{daughter}(X,Y); \text{son}(X,Y); \neg \text{parent}(Y,X) & . \ (15.8\%) \\
\text{parent}(Y,X); \neg \text{daughter}(X,Y) & . \ (8.5\%) \\
\text{parent}(Y,X); \neg \text{son}(X,Y) & . \ (7.1\%)
\end{align*}
\]

A third relevant pair of marginal frequencies (6 substitutions that make daughter(X,Y) true, 51 that make it false) could not be accommodated in this two-dimensional projection.
Between brackets the expected relative frequency of counter-instances is indicated, which we call the *confirmation* of the clause. For instance, the fourth clause has low confirmation because there are relatively few substitutions making $\text{son}(X,Y)$ true, and the same holds for $\text{daughter}(X,Y)$. 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 confirmation, 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. The approach is currently being implemented in the Primus\textsuperscript{4} system [13].

4 Concluding remarks

In this paper I have attempted to give an accessible introduction to Inductive Logic Programming. Both Horn clause induction and learning of integrity constraints have been reviewed. More detailed surveys are provided by [22, 30], while [24] offers an extensive overview of on-line available systems and datasets, as well as a bibliography with nearly 600 entries. [29, 4] are collections of research papers. http://www.ai.ijs.si/ilpnet.html is a good starting place for web searches.

Acknowledgements

An extended version of this paper, with emphasis on applications in Deductive Databases, will be published as [14]. This work was partially supported by ESPRIT IV Long Term Research Project 20237 Inductive Logic Programming 2.

References


\textsuperscript{4}Prime implicants uncovering system.


