Chapter 7

Inductive Logic Programming

There are many different representational forms for functions of input variables. So far, we have seen (Boolean) algebraic expressions, decision trees, and neural networks, plus other computational mechanisms such as techniques for computing nearest neighbors. Of course, the representation most important in computer science is a computer program. For example, a Lisp predicate of binary-valued inputs computes a Boolean function of those inputs. Similarly, a logic program (whose ordinary application is to compute bindings for variables) can also be used simply to decide whether or not a predicate has value *True (T)* or *False (F)*. For example, the Boolean exclusive-or (odd parity) function of two variables can be computed by the following logic program:

```
Parity(x,y) :- True(x), ¬ True(y)
            :- True(y), ¬ True(x)
```

We follow Prolog syntax (see, for example, [Mueller & Page, 1988]), except that our convention is to write variables as strings beginning with lower-case letters and predicates as strings beginning with upper-case letters. The unary function "`True`" returns T if and only if the value of its argument is T. (We now think of Boolean functions and arguments as having values of T and F instead of 0 and 1.) Programs will be written in "`typewriter`" font.

In this chapter, we consider the matter of learning logic programs given a set of variable values for which the logic program should return T (the *positive instances*) and a set of variable values for which it should return F (the *negative instances*). The subspecialty of machine learning that deals with learning logic programs is called *inductive logic programming (ILP)* [Lavrač & Džeroski, 1994]. As with any learning problem, this one can be quite complex and intractably difficult unless we constrain it with biases of some sort. In ILP, there are a variety of possible biases (called *language biases*). One might restrict the program to Horn clauses, not allow recursion, not allow functions, and so on.

As an example of an ILP problem, suppose we are trying to induce a function `Nonstop(x,y)`, that is to have value T for pairs of cities connected by a non-stop air flight and F for all other pairs of cities. We are given a training set consisting of positive and negative examples. As positive examples, we might have `(A,B)`, `(A, A1)`, and some other pairs; as negative examples, we might have `(A1, A2)`, and some other pairs. In ILP, we usually have additional information about the examples, called "background knowledge." In our air-flight problem, the background information might be such ground facts as `Hub(A)`, `Hub(B)`, `Satellite(A1,A)`, plus others. (`Hub(A)` is intended to mean that the city denoted by `A` is a hub city, and `Satellite(A1,A)` is intended to mean that the city denoted by `A1` is a satellite of the city denoted by `A`.) From these training facts, we want to induce a program `Nonstop(x,y)`, written in terms of the background relations `Hub` and `Satellite`, that has value T for all the positive instances and has value F for all the negative instances. Depending on the exact set of examples, we might induce the program:

```
Nonstop(x,y) :- Hub(x), Hub(y)

             :- Satellite(x,y)

             :- Satellite(y,x)
```

which would have value T if both of the two cities were hub cities or if one were a satellite of the other. As with other learning problems, we want the induced program to generalize well; that is, if presented with arguments not represented in the training set (but for which we have the needed background knowledge), we would like the function to guess well.

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

7.1 Notation and Definitions

In evaluating logic programs in ILP, we implicitly append the background facts to the program and adopt the usual convention that a program has value T for a set of inputs if and only if the program interpreter returns T when actually running the program (with background facts appended) on those inputs; otherwise it has value F. Using the given background facts, the program above would return T for input `(A, A1)`, for example. If a logic program, π, returns T for a set of arguments $\mathbf{X}$, we say that the program *covers* the arguments and write covers$(\pi, \mathbf{X})$. Following our terminology introduced in connection with version spaces, we will say that a program is *sufficient* if it covers all of the positive instances and that it is *necessary* if it does not cover any of the negative instances. (That is, a program implements a sufficient condition that a training instance is positive if it covers *all* of the positive training instances; it implements a necessary condition if it covers *none* of the negative instances.) In the noiseless case, we want to induce a program that is both sufficient and necessary, in which case we will call it *consistent*. With imperfect (noisy) training sets, we might relax this criterion and settle for a program that covers all but some fraction of the positive instances while allowing it to cover some fraction of the negative instances. We illustrate these definitions schematically in Fig. 7.1.

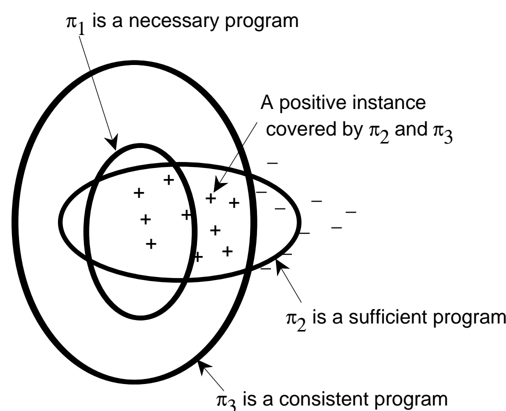

Figure 7.1: Sufficient, Necessary, and Consistent Programs

As in version spaces, if a program is sufficient but not necessary it can be

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

made to cover fewer examples by *specializing* it. Conversely, if it is necessary but not sufficient, it can be made to cover more examples by *generalizing* it. Suppose we are attempting to induce a logic program to compute the relation ρ. The most *general* logic program, which is certainly sufficient, is the one that has value T for *all* inputs, namely a single clause with an empty body, [ρ `:-`], which is called a *fact* in Prolog. The most *special* logic program, which is certainly necessary, is the one that has value F for *all* inputs, namely [ρ `:- F`]. Two of the many different ways to search for a consistent logic program are: 1) start with [ρ `:-`] and specialize until the program is consistent, or 2) start with [ρ `:- F`] and generalize until the program is consistent. We will be discussing a method that starts with [ρ `:-`], specializes until the program is necessary (but might no longer be sufficient), then reachieves sufficiency in stages by generalizing—ensuring within each stage that the program remains necessary (by specializing).

7.2 A Generic ILP Algorithm

Since the primary operators in our search for a consistent program are specialization and generalization, we must next discuss those operations. There are three major ways in which a logic program might be generalized:

1. Replace some terms in a program clause by variables. (Readers familiar with substitutions in the predicate calculus will note that this process is the inverse of substitution.)
2. Remove literals from the body of a clause.
3. Add a clause to the program

Analogously, there are three ways in which a logic program might be specialized:

1. Replace some variables in a program clause by terms (a *substitution*).
2. Add literals to the body of a clause.
3. Remove a clause from the program

We will be presenting an ILP learning method that adds clauses to a program when generalizing and that adds literals to the body of a clause when specializing. When we add a clause, we will always add the clause [ρ `:-`]

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

and then specialize it by adding literals to the body. Thus, we need only describe the process for adding literals.

Clauses can be partially ordered by the specialization relation. In general, clause c_1 is more special than clause c_2 if $c_2 \models c_1$. A special case, which is what we use here, is that a clause c_1 is more special than a clause c_2 if the set of literals in the body of c_2 is a subset of those in c_1. This ordering relation can be used in a structure of partially ordered clauses, called the *refinement graph*, that is similar to a version space. Clause c_1 is an immediate successor of clause c_2 in this graph if and only if clause c_1 can be obtained from clause c_2 by adding a literal to the body of c_2. A refinement graph then tells us the ways in which we can specialize a clause by adding a literal to it.

Of course there are unlimited possible literals we might add to the body of a clause. Practical ILP systems restrict the literals in various ways. Typical allowed additions are:

1. Literals used in the background knowledge.
2. Literals whose arguments are a subset of those in the head of the clause.
3. Literals that introduce a new distinct variable different from those in the head of the clause.
4. A literal that equates a variable in the head of the clause with another such variable or with a term mentioned in the background knowledge. (This possibility is equivalent to forming a specialization by making a substitution.)
5. A literal that is the same (except for its arguments) as that in the head of the clause. (This possibility admits recursive programs, which are disallowed in some systems.)

We can illustrate these possibilities using our air-flight example. We start with the program [`Nonstop(x,y) :-`]. The literals used in the background knowledge are `Hub` and `Satellite`. Thus the literals that we might consider adding are:

```
Hub(x)
Hub(y)
Hub(z)
Satellite(x,y)
```

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

```
Satellite(y,x)
Satellite(x,z)
Satellite(z,y)
(x = y)
```

(If recursive programs are allowed, we could also add the literals `Nonstop(x,z)` and `Nonstop(z,y)`.) These possibilities are among those illustrated in the refinement graph shown in Fig. 7.2. Whatever restrictions on additional literals are imposed, they are all syntactic ones from which the successors in the refinement graph are easily computed. ILP programs that follow the approach we are discussing (of specializing clauses by adding a literal) thus have well defined methods of computing the possible literals to add to a clause.

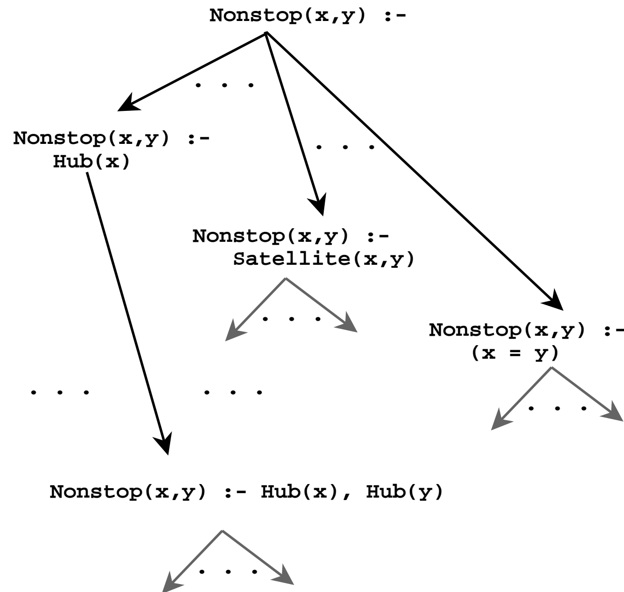

Figure 7.2: Part of a Refinement Graph

Now we are ready to write down a simple generic algorithm for inducing a logic program, π for inducing a relation ρ. We are given a training set, Ξ of argument sets some known to be in the relation ρ and some not in

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

ρ; Ξ^+ are the positive instances, and Ξ^- are the negative instances. The algorithm has an outer loop in which it successively adds clauses to make π more and more sufficient. It has an inner loop for constructing a clause, c, that is more and more necessary and in which it refers only to a subset, Ξ_{cur}, of the training instances. (The positive instances in Ξ_{cur} will be denoted by Ξ^+_{cur}, and the negative ones by Ξ^-_{cur}.) The algorithm is also given background relations and the means for adding literals to a clause. It uses a logic program interpreter to compute whether or not the program it is inducing covers training instances. The algorithm can be written as follows:

Generic ILP Algorithm

(Adapted from [Lavrač & Džeroski, 1994, p. 60].)

Initialize $\Xi_{cur} := \Xi$.
Initialize $\pi :=$ empty set of clauses.
repeat [The outer loop works to make π sufficient.]
    Initialize $c := \rho :-$.
    repeat [The inner loop makes c necessary.]
        Select a literal l to add to c. [This is a nondeterministic choice point.]
        Assign $c := c, l$.
    until c is necessary. [That is, until c covers no negative instances in Ξ_{cur}.]
    Assign $\pi := \pi, c$. [We add the clause c to the program.]
    Assign $\Xi_{cur} := \Xi_{cur} -$ (the positive instances in Ξ_{cur} covered by π).
until π is sufficient.

(The termination tests for the inner and outer loops can be relaxed as appropriate for the case of noisy instances.)

7.3 An Example

We illustrate how the algorithm works by returning to our example of airline flights. Consider the portion of an airline route map, shown in Fig. 7.3. Cities A, B, and C are "hub" cities, and we know that there are nonstop flights between all hub cities (even those not shown on this portion of the route map). The other cities are "satellites" of one of the hubs, and we know that there are nonstop flights between each satellite city and its hub. The learning program is given a set of positive instances, Ξ^+, of pairs of cities between which there are nonstop flights and a set of negative instances, Ξ^-,

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

of pairs of cities between which there are not nonstop flights. Ξ^+ contains just the pairs:

$$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$$
$$< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$$
$$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$$

For our example, we will assume that Ξ^- contains all those pairs of cities shown in Fig. 7.3 that are not in Ξ^+ (a type of *closed-world assumption*). These are:

$$\{< A, B1 >, < A, B2 >, < A, C1 >, < A, C2 >, < B, C1 >, < B, C2 >,$$
$$< B, A1 >, < B, A2 >, < C, A1 >, < C, A2 >, < C, B1 >, < C, B2 >,$$
$$< B1, A >, < B2, A >, < C1, A >, < C2, A >, < C1, B >, < C2, B >,$$
$$< A1, B >, < A2, B >, < A1, C >, < A2, C >, < B1, C >, < B2, C >\}$$

There may be other cities not shown on this map, so the training set does not necessarily exhaust all the cities.

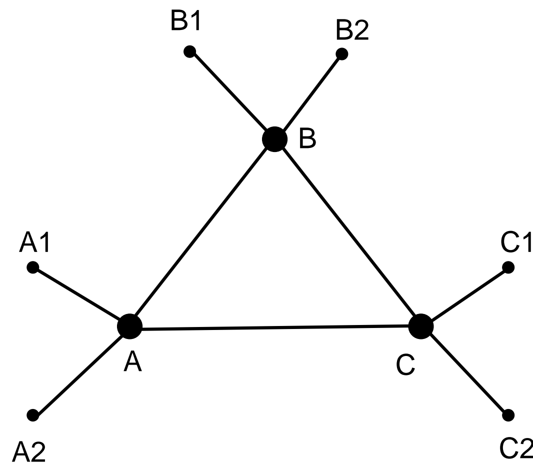

Figure 7.3: Part of an Airline Route Map

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

We want the learning program to induce a program for computing the value of the relation `Nonstop`. The training set, Ξ, can be thought of as a partial description of this relation in extensional form—it explicitly names some pairs in the relation and some pairs not in the relation. We desire to learn the `Nonstop` relation as a logic program in terms of the background relations, `Hub` and `Satellite`, which are also given in extensional form. Doing so will give us a more compact, *intensional*, description of the relation, and this description could well generalize usefully to other cities not mentioned in the map.

We assume the learning program has the following extensional definitions of the relations `Hub` and `Satellite`:

<u>`Hub`</u>

$$\{< A >, < B >, < C >\}$$

All other cities mentioned in the map are assumed not in the relation Hub. We will use the notation `Hub(x)` to express that the city named x is in the relation `Hub`.

<u>`Satellite`</u>

$$\{< A1, A, >, < A2, A >, < B1, B >, < B2, B >, < C1, C >, < C2, C >\}$$

All other pairs of cities mentioned in the map are not in the relation `Satellite`. We will use the notation `Satellite(x,y)` to express that the pair $< x, y >$ is in the relation `Satellite`.

Knowing that the predicate `Nonstop` is a two-place predicate, the inner loop of our algorithm initializes the first clause to `Nonstop(x,y) :-` . This clause is not necessary because it covers all the negative examples (since it covers all examples). So we must add a literal to its (empty) body. Suppose (selecting a literal from the refinement graph) the algorithm adds `Hub(x)`. The following positive instances in Ξ are covered by `Nonstop(x,y) :- Hub(x)`:

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$
$< A, A1 >, < A, A2 >, < B, B1 >, < B, B2 >, < C, C1 >, < C, C2 >\}$

To compute this covering, we interpret the logic program `Nonstop(x,y) :- Hub(x)` for all pairs of cities in Ξ, using the pairs given in the background relation `Hub` as ground facts. The following negative instances are also covered:

$\{< A, B1 >, < A, B2 >, < A, C1 >, < A, C2 >, < C, A1 >, < C, A2 >,$
$< C, B1 >, < C, B2 >, < B, A1 >, < B, A2 >, < B, C1 >, < B, C2 >\}$

Thus, the clause is not yet necessary and another literal must be added. Suppose we next add `Hub(y)`. The following positive instances are covered by `Nonstop(x,y) :- Hub(x), Hub(y)`:

$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >\}$

There are no longer any negative instances in Ξ covered so the clause `Nonstop(x,y) :- Hub(x), Hub(y)` is necessary, and we can terminate the first pass through the inner loop.

But the program, π, consisting of just this clause is not sufficient. These positive instances are *not* covered by the clause:

$\{< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$
$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$

The positive instances that were covered by `Nonstop(x,y) :- Hub(x), Hub(y)` are removed from Ξ to form the Ξ_{cur} to be used in the next pass through the inner loop. Ξ_{cur} consists of all the negative instances in Ξ plus the positive instances (listed above) that are not yet covered. In order to attempt to cover them, the inner loop creates another clause c, initially set to `Nonstop(x,y) :-` . This clause covers all the negative instances, and so we must add literals to make it necessary. Suppose we add the literal `Satellite(x,y)`. The clause `Nonstop(x,y) :- Satellite(x,y)` covers no negative instances, so it is necessary. It does cover the following positive instances in Ξ_{cur}:

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

$\{< A1, A >, < A2, A >, < B1, B >, < B2, B >, < C1, C >, < C2, C >\}$

These instances are removed from Ξ_{cur} for the next pass through the inner loop. The program now contains two clauses:

```
Nonstop(x,y) :- Hub(x), Hub(y)
             :- Satellite(x,y)
```

This program is not yet sufficient since it does not cover the following positive instances:

$\{< A, A1 >, < A, A2 >, < B, B1 >, < B, B2 >, < C, C1 >, < C, C2 >\}$

During the next pass through the inner loop, we add the clause `Nonstop(x,y) :- Satellite(y,x)`. This clause is necessary, and since the program containing all three clauses is now sufficient, the procedure terminates with:

```
Nonstop(x,y) :- Hub(x), Hub(y)
             :- Satellite(x,y)
             :- Satellite(y,x)
```

Since each clause is necessary, and the whole program is sufficient, the program is also consistent with all instances of the training set. Note that this program can be applied (perhaps with good generalization) to other cities besides those in our partial map—so long as we can evaluate the relations `Hub` and `Satellite` for these other cities. In the next section, we show how the technique can be extended to use recursion on the relation we are inducing. With that extension, the method can be used to induce more general logic programs.

7.4 Inducing Recursive Programs

To induce a recursive program, we allow the addition of a literal having the same predicate letter as that in the head of the clause. Various mechanisms must be used to ensure that such a program will terminate; one such is to make sure that the new literal has different variables than those in the

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

head literal. The process is best illustrated with another example. Our example continues the one using the airline map, but we make the map somewhat simpler in order to reduce the size of the extensional relations used. Consider the map shown in Fig. 7.4. Again, B and C are hub cities, *B1* and *B2* are satellites of B, *C1* and *C2* are satellites of C. We have introduced two new cities, *B3* and *C3*. No flights exist between these cities and any other cities—perhaps there are only bus routes as shown by the grey lines in the map.

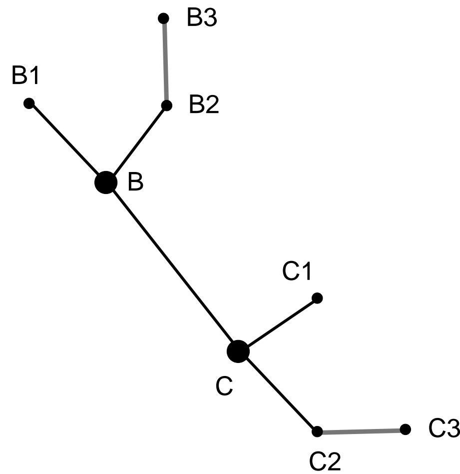

Figure 7.4: Another Airline Route Map

We now seek to learn a program for **Canfly(x,y)** that covers only those pairs of cities that can be reached by one or more nonstop flights. The relation **Canfly** is satisfied by the following pairs of postive instances:

$$\{< B1, B >, < B1, B2 >, < B1, C >, < B1, C1 >, < B1, C2 >,$$
$$< B, B1 >, < B2, B1 >, < C, B1 >, < C1, B1 >, < C2, B1 >,$$
$$< B2, B >, < B2, C >, < B2, C1 >, < B2, C2 >, < B, B2 >,$$
$$< C, B2 >, < C1, B2 >, < C2, B2 >, < B, C >, < B, C1 >,$$
$$< B, C2 >, < C, B >, < C1, B >, < C2, B >, < C, C1 >,$$
$$< C, C2 >, < C1, C >, < C2, C >, < C1, C2 >, < C2, C1 >\}$$

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

Using a closed-world assumption on our map, we take the negative instances of `Canfly` to be:

$$\{< B3, B2 >, < B3, B >, < B3, B1 >, < B3, C >, < B3, C1 >, < B3, C2 >, < B3, C3 >, < B2, B3 >, < B, B3 >, < B1, B3 >, < C, B3 >, < C1, B3 >, < C2, B3 >, < C3, B3 >, < C3, B2 >, < C3, B >, < C3, B1 >, < C3, C >, < C3, C1 >, < C3, C2 >, < B2, C3 >, < B, C3 >, < B1, C3 >, < C, C3 >, < C1, C3 >, < C2, C3 >\}$$

We will induce `Canfly(x,y)` using the extensionally defined background relation `Nonstop` given earlier (modified as required for our reduced airline map) and `Canfly` itself (recursively).

As before, we start with the empty program and proceed to the inner loop to construct a clause that is necessary. Suppose that the inner loop adds the background literal `Nonstop(x,y)`. The clause `Canfly(x,y) :- Nonstop(x,y)` is necessary; it covers no negative instances. But it is not sufficient because it does not cover the following positive instances:

$$\{< B1, B2 >, < B1, C >, < B1, C1 >, < B1, C2 >, < B2, B1 >, < C, B1 >, < C1, B1 >, < C2, B1 >, < B2, C >, < B2, C1 >, < B2, C2 >, < C, B2 >, < C1, B2 >, < C2, B2 >, < B, C1 >, < B, C2 >, < C1, B >, < C2, B >, < C1, C2 >, < C2, C1 >\}$$

Thus, we must add another clause to the program. In the inner loop, we first create the clause `Canfly(x,y) :- Nonstop(x,z)` which introduces the new variable z. We digress briefly to describe how a program containing a clause with unbound variables in its body is interpreted. Suppose we try to interpret it for the positive instance `Canfly(B1,B2)`. The interpreter attempts to establish `Nonstop(B1,z)` for some z. Since `Nonstop(B1, B)`, for example, is a background fact, the interpreter returns T—which means that the instance $< B1, B2 >$ is covered. Suppose now, we attempt to interpret the clause for the negative instance `Canfly(B3,B)`. The interpreter attempts to establish `Nonstop(B3,z)` for some z. There are no background facts that match, so the clause does not cover $< B3, B >$. Using the interpreter, we see that the clause `Canfly(x,y) :- Nonstop(x,z)` covers all of the positive instances not already covered by the first clause, but it also covers many negative instances such as $< B2, B3 >$, and $< B, B3 >$. So the inner

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

loop must add another literal. This time, suppose it adds `Canfly(y,z)` to yield the clause `Canfly(x,y) :- Nonstop(x,z), Canfly(y,z)`. This clause is necessary; no negative instances are covered. The program is now sufficient and consistent; it is:

```
Canfly(x,y) :- Nonstop(x,y)
            :- Nonstop(x,z), Canfly(y,z)
```

7.5 Choosing Literals to Add

One of the first practical ILP systems was Quinlan's FOIL [Quinlan, 1990]. A major problem involves deciding how to select a literal to add in the inner loop (from among the literals that are allowed). In FOIL, Quinlan suggested that candidate literals can be compared using an information-like measure—similar to the measures used in inducing decision trees. A measure that gives the same comparison as does Quinlan's is based on the amount by which adding a literal increases the *odds* that an instance drawn at random from those covered by the new clause is a positive instance beyond what these odds were before adding the literal.

Let p be an estimate of the probability that an instance drawn at random from those covered by a clause before adding the literal is a positive instance. That is, p =(number of positive instances covered by the clause)/(total number of instances covered by the clause). It is convenient to express this probability in "odds form." The odds, o, that a covered instance is positive is defined to be $o = p/(1-p)$. Expressing the probability in terms of the odds, we obtain $p = o/(1+o)$.

After selecting a literal, l, to add to a clause, some of the instances previously covered are still covered; some of these are positive and some are negative. Let p_l denote the probability that an instance drawn at random from the instances covered by the new clause (with l added) is positive. The odds will be denoted by o_l. We want to select a literal, l, that gives maximal increase in these odds. That is, if we define $\lambda_l = o_l/o$, we want a literal that gives a high value of λ_l. Specializing the clause in such a way that it fails to cover many of the negative instances previously covered but still covers most of the positive instances previously covered will result in a high value of λ_l. (It turns out that the value of Quinlan's information theoretic measure increases monotonically with λ_l, so we could just as well use the latter instead.)

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

Besides finding a literal with a high value of λ_l, Quinlan's FOIL system also restricts the choice to literals that:

a) contain at least one variable that has already been used,

b) place further restrictions on the variables if the literal selected has the same predicate letter as the literal being induced (in order to prevent infinite recursion), and

c) survive a pruning test based on the values of λ_l for those literals selected so far.

We refer the reader to Quinlan's paper for further discussion of these points. Quinlan also discusses post-processing pruning methods and presents experimental results of the method applied to learning recursive relations on lists, on learning rules for chess endgames and for the card game Eleusis, and for some other standard tasks mentioned in the machine learning literature.

The reader should also refer to [Pazzani & Kibler, 1992, Lavrač & Džeroski, 1994, Muggleton, 1991, Muggleton, 1992].

Discuss preprocessing, postprocessing, bottom-up methods, and LINUS.

7.6 Relationships Between ILP and Decision Tree Induction

The generic ILP algorithm can also be understood as a type of decision tree induction. Recall the problem of inducing decision trees when the values of attributes are categorical. When splitting on a single variable, the split at each node involves asking to which of several mutually exclusive and exhaustive subsets the value of a variable belongs. For example, if a node tested the variable x_i, and if x_i could have values drawn from $\{A, B, C, D, E, F\}$, then one possible split (among many) might be according to whether the value of x_i had as value one of $\{A, B, C\}$ or one of $\{D, E, F\}$.

It is also possible to make a multi-variate split—testing the values of two or more variables at a time. With categorical variables, an n-variable split would be based on which of several n-ary relations the values of the variables satisfied. For example, if a node tested the variables x_i and x_j, and if x_i and x_j both could have values drawn from $\{A, B, C, D, E, F\}$, then one possible binary split (among many) might be according to whether or not $< x_i, x_j >$ satisfied the relation $\{< A, C >, < C, D >\}$. (Note that our subset method of forming single-variable splits could equivalently have been framed using 1-ary relations—which are usually called properties.)

In this framework, the ILP problem is as follows: We are given a training set, Ξ, of positively and negatively labeled patterns whose components are

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

drawn from a set of variables $\{x, y, z, \ldots\}$. The positively labeled patterns in Ξ form an extensional definition of a relation, R. We are also given background relations, $R_1, \ldots, R_k$, on various subsets of these variables. (That is, we are given sets of tuples that are in these relations.) We desire to construct an intensional definition of R in terms of the $R_1, \ldots, R_k$, such that all of the positively labeled patterns in Ξ are satisfied by R and none of the negatively labeled patterns are. The intensional definition will be in terms of a logic program in which the relation R is the head of a set of clauses whose bodies involve the background relations.

The generic ILP algorithm can be understood as decision tree induction, where each node of the decision tree is itself a sub-decision tree, and each sub-decision tree consists of nodes that make binary splits on several variables using the background relations, R_i. Thus we will speak of a top-level decision tree and various sub-decision trees. (Actually, our decision trees will be decision lists—a special case of decision trees, but we will refer to them as trees in our discussions.)

In broad outline, the method for inducing an intensional version of the relation R is illustrated by considering the decision tree shown in Fig. 7.5. In this diagram, the patterns in Ξ are first filtered through the decision tree in top-level node 1. The background relation R_1 is satisfied by some of these patterns; these are filtered to the right (to relation R_2), and the rest are filtered to the left (more on what happens to these later). Right-going patterns are filtered through a sequence of relational tests until only positively labeled patterns satisfy the last relation—in this case R_3. That is, the subset of patterns satisfying all the relations, R_1, R_2, and R_3 contains only positive instances from Ξ. (We might say that this combination of tests is necessary. They correspond to the clause created in the first pass through the inner loop of the generic ILP algorithm.) Let us call the subset of patterns satisfying these relations, Ξ_1; these satisfy Node 1 at the top level. All other patterns, that is $\{\Xi - \Xi_1\} = \Xi_2$ are filtered to the left by Node 1.

Ξ_2 is then filtered by top-level Node 2 in much the same manner, so that Node 2 is satisfied only by the positively labeled samples in Ξ_2. We continue filtering through top-level nodes until only the negatively labeled patterns fail to satisfy a top node. In our example, Ξ_4 contains only negatively labeled patterns and the union of Ξ_1 and Ξ_3 contains all the positively labeled patterns. The relation, R, that distinguishes positive from negative patterns in Ξ is then given in terms of the following logic program:

```
R :- R1, R2, R3
  :- R4, R5
```

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

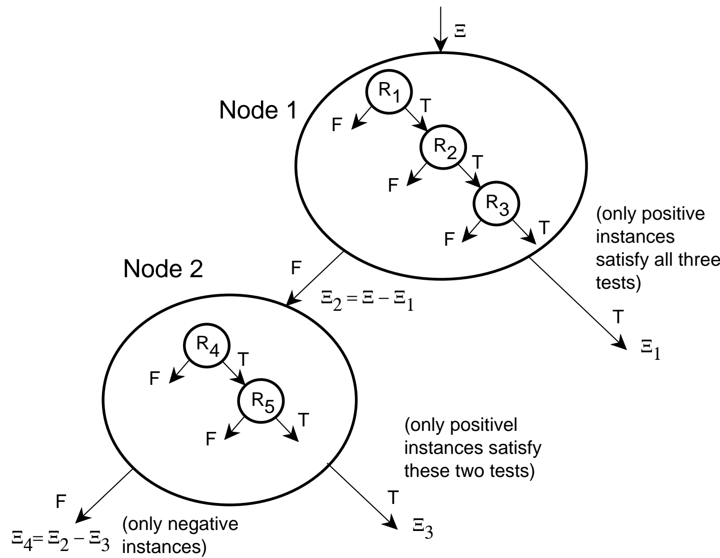

Figure 7.5: A Decision Tree for ILP

If we apply this sort of decision-tree induction procedure to the problem of generating a logic program for the relation `Nonstop` (refer to Fig. 7.3), we obtain the decision tree shown in Fig. 7.6. The logic program resulting from this decision tree is the same as that produced by the generic ILP algorithm.

In setting up the problem, the training set, Ξ can be expressed as a set of 2-dimensional vectors with components x and y. The values of these components range over the cities $\{A, B, C, A1, A2, B1, B2, C1, C2\}$ except (for simplicity) we do not allow patterns in which x and y have the same value. As before, the relation, `Nonstop`, contains the following pairs of cities, which are the positive instances:

$$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$$
$$< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$$
$$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$$

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

All other pairs of cities named in the map of Fig. 7.3 (using the closed world assumption) are not in the relation **Nonstop** and thus are negative instances.

Because the values of x and y are categorical, decision-tree induction would be a very difficult task—involving as it does the need to invent relations on x and y to be used as tests. But with the background relations, R_i (in this case **Hub** and **Satellite**), the problem is made much easier. We select these relations in the same way that we select literals; from among the available tests, we make a selection based on which leads to the largest value of λ_{R_i}.

7.7 Bibliographical and Historical Remarks

To be added.

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

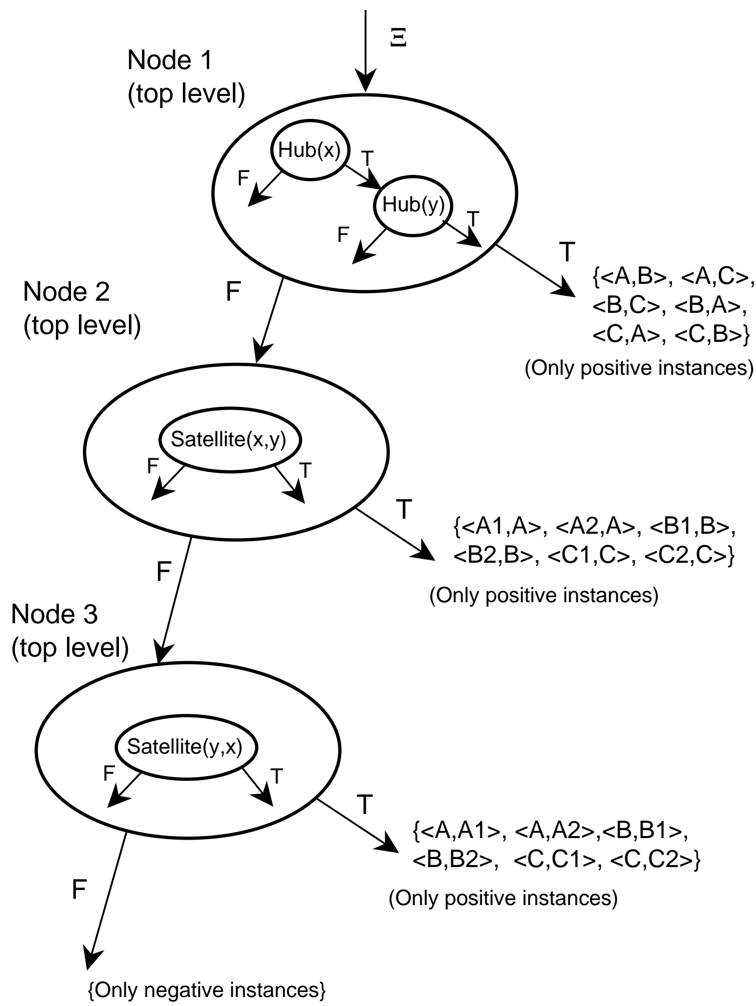

Figure 7.6: A Decision Tree for the Airline Route Problem

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.

Introduction to Machine Learning ©1996 Nils J. Nilsson. All rights reserved.