CHAPTER 10

LEARNING SETS OF RULES

One of the most expressive and human readable representations for learned hypotheses is sets of if-then rules. This chapter explores several algorithms for learning such sets of rules. One important special case involves learning sets of rules containing variables, called first-order Horn clauses. Because sets of first-order Horn clauses can be interpreted as programs in the logic programming language PROLOG, learning them is often called inductive logic programming (ILP). This chapter examines several approaches to learning sets of rules, including an approach based on inverting the deductive operators of mechanical theorem provers.

10.1 INTRODUCTION

In many cases it is useful to learn the target function represented as a set of if-then rules that jointly define the function. As shown in Chapter 3, one way to learn sets of rules is to first learn a decision tree, then translate the tree into an equivalent set of rules—one rule for each leaf node in the tree. A second method, illustrated in Chapter 9, is to use a genetic algorithm that encodes each rule set as a bit string and uses genetic search operators to explore this hypothesis space. In this chapter we explore a variety of algorithms that directly learn rule sets and that differ from these algorithms in two key respects. First, they are designed to learn sets of first-order rules that contain variables. This is significant because first-order rules are much more expressive than propositional rules. Second, the algorithms discussed here use sequential covering algorithms that learn one rule at a time to incrementally grow the final set of rules.

As an example of first-order rule sets, consider the following two rules that jointly describe the target concept *Ancestor*. Here we use the predicate $Parent(x, y)$ to indicate that y is the mother or father of x, and the predicate $Ancestor(x, y)$ to indicate that y is an ancestor of x related by an arbitrary number of family generations.

IF $Parent(x, y)$ THEN $Ancestor(x, y)$
IF $Parent(x, z) \wedge Ancestor(z, y)$ THEN $Ancestor(x, y)$

Note these two rules compactly describe a recursive function that would be very difficult to represent using a decision tree or other propositional representation. One way to see the representational power of first-order rules is to consider the general purpose programming language PROLOG. In PROLOG, programs are sets of first-order rules such as the two shown above (rules of this form are also called *Horn clauses*). In fact, when stated in a slightly different syntax the above rules form a valid PROLOG program for computing the *Ancestor* relation. In this light, a general purpose algorithm capable of learning such rule sets may be viewed as an algorithm for automatically inferring PROLOG programs from examples. In this chapter we explore learning algorithms capable of learning such rules, given appropriate sets of training examples.

In practice, learning systems based on first-order representations have been successfully applied to problems such as learning which chemical bonds fragment in a mass spectrometer (Buchanan 1976; Lindsay 1980), learning which chemical substructures produce mutagenic activity (a property related to carcinogenicity) (Srinivasan et al. 1994), and learning to design finite element meshes to analyze stresses in physical structures (Dolsak and Muggleton 1992). In each of these applications, the hypotheses that must be represented involve relational assertions that can be conveniently expressed using first-order representations, while they are very difficult to describe using propositional representations.

In this chapter we begin by considering algorithms that learn sets of propositional rules; that is, rules without variables. Algorithms for searching the hypothesis space to learn disjunctive sets of rules are most easily understood in this setting. We then consider extensions of these algorithms to learn first-order rules. Two general approaches to inductive logic programming are then considered, and the fundamental relationship between inductive and deductive inference is explored.

10.2 SEQUENTIAL COVERING ALGORITHMS

Here we consider a family of algorithms for learning rule sets based on the strategy of learning one rule, removing the data it covers, then iterating this process. Such algorithms are called *sequential covering* algorithms. To elaborate, imagine we have a subroutine LEARN-ONE-RULE that accepts a set of positive and negative training examples as input, then outputs a single rule that covers many of the

positive examples and few of the negative examples. We require that this output rule have high accuracy, but not necessarily high coverage. By high accuracy, we mean the predictions it makes should be correct. By accepting low coverage, we mean it need not make predictions for every training example.

Given this LEARN-ONE-RULE subroutine for learning a single rule, one obvious approach to learning a set of rules is to invoke LEARN-ONE-RULE on all the available training examples, remove any positive examples covered by the rule it learns, then invoke it again to learn a second rule based on the remaining training examples. This procedure can be iterated as many times as desired to learn a disjunctive set of rules that together cover any desired fraction of the positive examples. This is called a *sequential covering* algorithm because it sequentially learns a set of rules that together cover the full set of positive examples. The final set of rules can then be sorted so that more accurate rules will be considered first when a new instance must be classified. A prototypical sequential covering algorithm is described in Table 10.1.

This sequential covering algorithm is one of the most widespread approaches to learning disjunctive sets of rules. It reduces the problem of learning a disjunctive set of rules to a sequence of simpler problems, each requiring that a single conjunctive rule be learned. Because it performs a greedy search, formulating a sequence of rules without backtracking, it is not guaranteed to find the smallest or best set of rules that cover the training examples.

How shall we design LEARN-ONE-RULE to meet the needs of the sequential covering algorithm? We require an algorithm that can formulate a single rule with high accuracy, but that need not cover all of the positive examples. In this section we present a variety of algorithms and describe the main variations that have been explored in the research literature. In this section we consider learning only propositional rules. In later sections, we extend these algorithms to learn first-order Horn clauses.

SEQUENTIAL-COVERING(*Target_attribute*, *Attributes*, *Examples*, *Threshold*)

- *Learned_rules* ← {}
- *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- while PERFORMANCE(*Rule*, *Examples*) > *Threshold*, do
  - *Learned_rules* ← *Learned_rules* + *Rule*
  - *Examples* ← *Examples* − {examples correctly classified by *Rule*}
  - *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- *Learned_rules* ← sort *Learned_rules* accord to PERFORMANCE over *Examples*
- return *Learned_rules*

TABLE 10.1
The sequential covering algorithm for learning a disjunctive set of rules. LEARN-ONE-RULE must return a single rule that covers at least some of the *Examples*. PERFORMANCE is a user-provided subroutine to evaluate rule quality. This covering algorithm learns rules until it can no longer learn a rule whose performance is above the given *Threshold*.

10.2.1 General to Specific Beam Search

One effective approach to implementing LEARN-ONE-RULE is to organize the hypothesis space search in the same general fashion as the ID3 algorithm, but to follow only the most promising branch in the tree at each step. As illustrated in the search tree of Figure 10.1, the search begins by considering the most general rule precondition possible (the empty test that matches every instance), then greedily adding the attribute test that most improves rule performance measured over the training examples. Once this test has been added, the process is repeated by greedily adding a second attribute test, and so on. Like ID3, this process grows the hypothesis by greedily adding new attribute tests until the hypothesis reaches an acceptable level of performance. Unlike ID3, this implementation of LEARN-ONE-RULE follows only a single descendant at each search step—the attribute-value pair yielding the best performance—rather than growing a subtree that covers all possible values of the selected attribute.

This approach to implementing LEARN-ONE-RULE performs a general-to-specific search through the space of possible rules in search of a rule with high accuracy, though perhaps incomplete coverage of the data. As in decision tree learning, there are many ways to define a measure to select the "best" descendant. To follow the lead of ID3 let us for now define the best descendant as the one whose covered examples have the lowest entropy (recall Equation [3.3]).

The general-to-specific search suggested above for the LEARN-ONE-RULE algorithm is a greedy depth-first search with no backtracking. As with any greedy

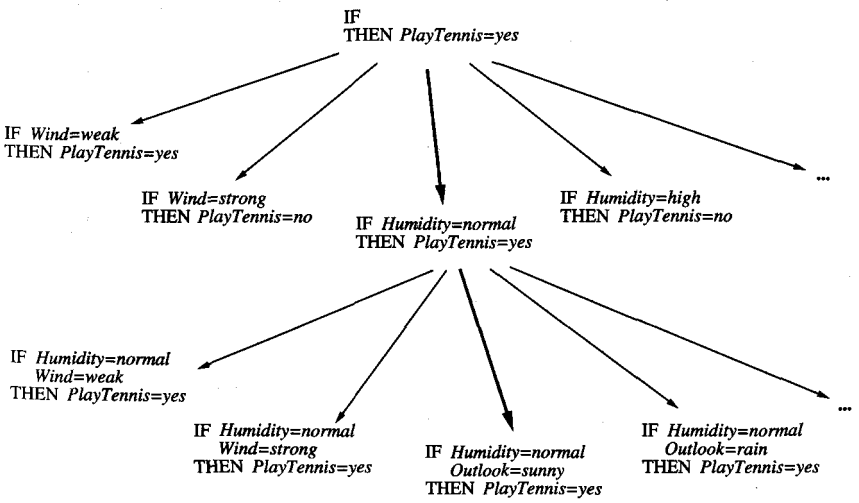

FIGURE 10.1
The search for rule preconditions as LEARN-ONE-RULE proceeds from general to specific. At each step, the preconditions of the best rule are specialized in all possible ways. Rule postconditions are determined by the examples found to satisfy the preconditions. This figure illustrates a beam search of width 1.

search, there is a danger that a suboptimal choice will be made at any step. To reduce this risk, we can extend the algorithm to perform a *beam search*; that is, a search in which the algorithm maintains a list of the k best candidates at each step, rather than a single best candidate. On each search step, descendants (specializations) are generated for each of these k best candidates, and the resulting set is again reduced to the k most promising members. Beam search keeps track of the most promising alternatives to the current top-rated hypothesis, so that all of their successors can be considered at each search step. This general to specific beam search algorithm is used by the CN2 program described by Clark and Niblett (1989). The algorithm is described in Table 10.2.

LEARN-ONE-RULE(*Target_attribute, Attributes, Examples, k*)

Returns a single rule that covers some of the Examples. Conducts a general_to_specific greedy beam search for the best rule, guided by the PERFORMANCE *metric.*

- Initialize *Best_hypothesis* to the most general hypothesis $\emptyset$
- Initialize *Candidate_hypotheses* to the set {*Best_hypothesis*}
- While *Candidate_hypotheses* is not empty, Do
  1. *Generate the next more specific candidate_hypotheses*
     - *All_constraints* ← the set of all constraints of the form $(a = v)$, where a is a member of *Attributes*, and v is a value of a that occurs in the current set of *Examples*
     - *New_candidate_hypotheses* ←
       for each h in *Candidate_hypotheses*,
       for each c in *All_constraints*,
       - create a specialization of h by adding the constraint c
     - Remove from *New_candidate_hypotheses* any hypotheses that are duplicates, inconsistent, or not maximally specific
  2. *Update Best_hypothesis*
     - For all h in *New_candidate_hypotheses* do
       - If (PERFORMANCE(h, *Examples*, *Target_attribute*)
         > PERFORMANCE(*Best_hypothesis*, *Examples*, *Target_attribute*))
         Then *Best_hypothesis* ← h
  3. *Update Candidate_hypotheses*
     - *Candidate_hypotheses* ← the k best members of *New_candidate_hypotheses*, according to the PERFORMANCE measure.
- Return a rule of the form
  "IF *Best_hypothesis* THEN *prediction*"
  where *prediction* is the most frequent value of *Target_attribute* among those *Examples* that match *Best_hypothesis*.

PERFORMANCE(h, *Examples*, *Target_attribute*)

- *h_examples* ← the subset of *Examples* that match h
- return $-Entropy(h\_examples)$, where entropy is with respect to *Target_attribute*

TABLE 10.2
One implementation for LEARN-ONE-RULE is a general-to-specific beam search. The frontier of current hypotheses is represented by the variable *Candidate_hypotheses*. This algorithm is similar to that used by the CN2 program, described by Clark and Niblett (1989).

A few remarks on the LEARN-ONE-RULE algorithm of Table 10.2 are in order. First, note that each hypothesis considered in the main loop of the algorithm is a conjunction of attribute-value constraints. Each of these conjunctive hypotheses corresponds to a candidate set of preconditions for the rule to be learned and is evaluated by the entropy of the examples it covers. The search considers increasingly specific candidate hypotheses until it reaches a maximally specific hypothesis that contains all available attributes. The rule that is output by the algorithm is the rule encountered during the search whose PERFORMANCE is greatest—not necessarily the final hypothesis generated in the search. The postcondition for the output rule is chosen only in the final step of the algorithm, after its precondition (represented by the variable *Best_hypothesis*) has been determined. The algorithm constructs the rule postcondition to predict the value of the target attribute that is most common among the examples covered by the rule precondition. Finally, note that despite the use of beam search to reduce the risk, the greedy search may still produce suboptimal rules. However, even when this occurs the SEQUENTIAL-COVERING algorithm can still learn a collection of rules that together cover the training examples, because it repeatedly calls LEARN-ONE-RULE on the remaining uncovered examples.

10.2.2 Variations

The SEQUENTIAL-COVERING algorithm, together with the LEARN-ONE-RULE algorithm, learns a set of if-then rules that covers the training examples. Many variations on this approach have been explored. For example, in some cases it might be desirable to have the program learn only rules that cover positive examples and to include a "default" that assigns a negative classification to instances not covered by any rule. This approach might be desirable, say, if one is attempting to learn a target concept such as "pregnant women who are likely to have twins." In this case, the fraction of positive examples in the entire population is small, so the rule set will be more compact and intelligible to humans if it identifies only classes of positive examples, with the default classification of all other examples as negative. This approach also corresponds to the "negation-as-failure" strategy of PROLOG, in which any expression that cannot be proven to be true is by default assumed to be false. In order to learn such rules that predict just a single target value, the LEARN-ONE-RULE algorithm can be modified to accept an additional input argument specifying the target value of interest. The general-to-specific beam search is conducted just as before, changing only the PERFORMANCE subroutine that evaluates hypotheses. Note the definition of PERFORMANCE as negative entropy is no longer appropriate in this new setting, because it assigns a maximal score to hypotheses that cover exclusively negative examples, as well as those that cover exclusively positive examples. Using a measure that evaluates the fraction of positive examples covered by the hypothesis would be more appropriate in this case.

Another variation is provided by a family of algorithms called AQ (Michalski 1969, Michalski et al. 1986), that predate the CN2 algorithm on which the

above discussion is based. Like CN2, AQ learns a disjunctive set of rules that together cover the target function. However, AQ differs in several ways from the algorithms given here. First, the covering algorithm of AQ differs from the SEQUENTIAL-COVERING algorithm because it explicitly seeks rules that cover a particular target value, learning a disjunctive set of rules for each target value in turn. Second, AQ's algorithm for learning a single rule differs from LEARN-ONE-RULE. While it conducts a general-to-specific beam search for each rule, it uses a single positive example to focus this search. In particular, it considers only those attributes satisfied by the positive example as it searches for progressively more specific hypotheses. Each time it learns a new rule it selects a new positive example from those that are not yet covered, to act as a seed to guide the search for this new disjunct.

10.3 LEARNING RULE SETS: SUMMARY

The SEQUENTIAL-COVERING algorithm described above and the decision tree learning algorithms of Chapter 3 suggest a variety of possible methods for learning sets of rules. This section considers several key dimensions in the design space of such rule learning algorithms.

First, *sequential covering* algorithms learn one rule at a time, removing the covered examples and repeating the process on the remaining examples. In contrast, decision tree algorithms such as ID3 learn the entire set of disjuncts simultaneously as part of the single search for an acceptable decision tree. We might, therefore, call algorithms such as ID3 *simultaneous covering* algorithms, in contrast to sequential covering algorithms such as CN2. Which should we prefer? The key difference occurs in the choice made at the most primitive step in the search. At each search step ID3 chooses among alternative *attributes* by comparing the *partitions* of the data they generate. In contrast, CN2 chooses among alternative *attribute-value* pairs, by comparing the *subsets* of data they cover. One way to see the significance of this difference is to compare the number of distinct choices made by the two algorithms in order to learn the same set of rules. To learn a set of n rules, each containing k attribute-value tests in their preconditions, sequential covering algorithms will perform $n \cdot k$ primitive search steps, making an independent decision to select each precondition of each rule. In contrast, simultaneous covering algorithms will make many fewer independent choices, because each choice of a decision node in the decision tree corresponds to choosing the precondition for the multiple rules associated with that node. In other words, if the decision node tests an attribute that has m possible values, the choice of the decision node corresponds to choosing a precondition for each of the m corresponding rules (see Exercise 10.1). Thus, sequential covering algorithms such as CN2 make a larger number of independent choices than simultaneous covering algorithms such as ID3. Still, the question remains, which should we prefer? The answer may depend on how much training data is available. If data is plentiful, then it may support the larger number of independent decisions required by the sequential covering algorithm, whereas if data is scarce, the "sharing" of

decisions regarding preconditions of different rules may be more effective. An additional consideration is the task-specific question of whether it is desirable that different rules test the same attributes. In the simultaneous covering decision tree learning algorithms, they will. In sequential covering algorithms, they need not.

A second dimension along which approaches vary is the direction of the search in LEARN-ONE-RULE. In the algorithm described above, the search is from *general to specific* hypotheses. Other algorithms we have discussed (e.g., FIND-S from Chapter 2) search from *specific to general.* One advantage of general to specific search in this case is that there is a single maximally general hypothesis from which to begin the search, whereas there are very many specific hypotheses in most hypothesis spaces (i.e., one for each possible instance). Given many maximally specific hypotheses, it is unclear which to select as the starting point of the search. One program that conducts a specific-to-general search, called GOLEM (Muggleton and Feng 1990), addresses this issue by choosing several positive examples at random to initialize and to guide the search. The best hypothesis obtained through multiple random choices is then selected.

A third dimension is whether the LEARN-ONE-RULE search is a *generate then test* search through the syntactically legal hypotheses, as it is in our suggested implementation, or whether it is *example-driven* so that individual training examples constrain the generation of hypotheses. Prototypical example-driven search algorithms include the FIND-S and CANDIDATE-ELIMINATION algorithms of Chapter 2, the AQ algorithm, and the CIGOL algorithm discussed later in this chapter. In each of these algorithms, the generation or revision of hypotheses is driven by the analysis of an individual training example, and the result is a revised hypothesis designed to correct performance for this single example. This contrasts to the generate and test search of LEARN-ONE-RULE in Table 10.2, in which successor hypotheses are generated based only on the syntax of the hypothesis representation. The training data is considered only after these candidate hypotheses are generated and is used to choose among the candidates based on their performance over the entire collection of training examples. One important advantage of the generate and test approach is that each choice in the search is based on the hypothesis performance over *many* examples, so that the impact of noisy data is minimized. In contrast, example-driven algorithms that refine the hypothesis based on individual examples are more easily misled by a single noisy training example and are therefore less robust to errors in the training data.

A fourth dimension is whether and how rules are post-pruned. As in decision tree learning, it is possible for LEARN-ONE-RULE to formulate rules that perform very well on the training data, but less well on subsequent data. As in decision tree learning, one way to address this issue is to post-prune each rule after it is learned from the training data. In particular, preconditions can be removed from the rule whenever this leads to improved performance over a set of pruning examples distinct from the training examples. A more detailed discussion of rule post-pruning is provided in Section 3.7.1.2.

A final dimension is the particular definition of rule PERFORMANCE used to guide the search in LEARN-ONE-RULE. Various evaluation functions have been used. Some common evaluation functions include:

- *Relative frequency.* Let n denote the number of examples the rule matches and let n_c denote the number of these that it classifies correctly. The relative frequency estimate of rule performance is

$$\frac{n_c}{n}$$

  Relative frequency is used to evaluate rules in the AQ program.

- *m-estimate of accuracy.* This accuracy estimate is biased toward the default accuracy expected of the rule. It is often preferred when data is scarce and the rule must be evaluated based on few examples. As above, let n and n_c denote the number of examples matched and correctly predicted by the rule. Let p be the prior probability that a randomly drawn example from the entire data set will have the classification assigned by the rule (e.g., if 12 out of 100 examples have the value predicted by the rule, then $p = .12$). Finally, let m be the weight, or equivalent number of examples for weighting this prior p. The m-estimate of rule accuracy is

$$\frac{n_c + mp}{n + m}$$

  Note if m is set to zero, then the m-estimate becomes the above relative frequency estimate. As m is increased, a larger number of examples is needed to override the prior assumed accuracy p. The m-estimate measure is advocated by Cestnik and Bratko (1991) and has been used in some versions of the CN2 algorithm. It is also used in the naive Bayes classifier discussed in Section 6.9.1.

- *Entropy.* This is the measure used by the PERFORMANCE subroutine in the algorithm of Table 10.2. Let S be the set of examples that match the rule preconditions. Entropy measures the uniformity of the target function values for this set of examples. We take the negative of the entropy so that better rules will have higher scores.

$$-Entropy(S) = \sum_{i=1}^{c} p_i \log_2 p_i$$

  where c is the number of distinct values the target function may take on, and where p_i is the proportion of examples from S for which the target function takes on the ith value. This entropy measure, combined with a test for statistical significance, is used in the CN2 algorithm of Clark and Niblett (1989). It is also the basis for the information gain measure used by many decision tree learning algorithms.

10.4 LEARNING FIRST-ORDER RULES

In the previous sections we discussed algorithms for learning sets of propositional (i.e., variable-free) rules. In this section, we consider learning rules that contain variables—in particular, learning first-order Horn theories. Our motivation for considering such rules is that they are much more expressive than propositional rules. Inductive learning of first-order rules or theories is often referred to as *inductive logic programming* (or ILP for short), because this process can be viewed as automatically inferring PROLOG programs from examples. PROLOG is a general purpose, Turing-equivalent programming language in which programs are expressed as collections of Horn clauses.

10.4.1 First-Order Horn Clauses

To see the advantages of first-order representations over propositional (variable-free) representations, consider the task of learning the simple target concept $Daughter(x, y)$, defined over pairs of people x and y. The value of $Daughter(x, y)$ is $True$ when x is the daughter of y, and $False$ otherwise. Suppose each person in the data is described by the attributes $Name$, $Mother$, $Father$, $Male$, $Female$. Hence, each training example will consist of the description of two people in terms of these attributes, along with the value of the target attribute $Daughter$. For example, the following is a positive example in which Sharon is the daughter of Bob:

$$\begin{array}{lll} \langle Name_1 = Sharon, & Mother_1 = Louise, & Father_1 = Bob, \\ Male_1 = False, & Female_1 = True, & \\ Name_2 = Bob, & Mother_2 = Nora, & Father_2 = Victor, \\ Male_2 = True, & Female_2 = False, & Daughter_{1,2} = True\rangle \end{array}$$

where the subscript on each attribute name indicates which of the two persons is being described. Now if we were to collect a number of such training examples for the target concept $Daughter_{1,2}$ and provide them to a propositional rule learner such as CN2 or C4.5, the result would be a collection of very specific rules such as

$$\begin{array}{ll} \text{IF} & (Father_1 = Bob) \wedge (Name_2 = Bob) \wedge (Female_1 = True) \\ \text{THEN} & Daughter_{1,2} = True \end{array}$$

Although it is correct, this rule is so specific that it will rarely, if ever, be useful in classifying future pairs of people. The problem is that propositional representations offer no general way to describe the essential *relations* among the values of the attributes. In contrast, a program using first-order representations could learn the following general rule:

$$\text{IF} \quad Father(y, x) \wedge Female(y), \quad \text{THEN} \quad Daughter(x, y)$$

where x and y are variables that can be bound to any person.

First-order Horn clauses may also refer to variables in the preconditions that do not occur in the postconditions. For example, one rule for *GrandDaughter* might be

IF $Father(y, z) \wedge Mother(z, x) \wedge Female(y)$
THEN $GrandDaughter(x, y)$

Note the variable z in this rule, which refers to the father of y, is not present in the rule postconditions. Whenever such a variable occurs only in the preconditions, it is assumed to be existentially quantified; that is, the rule preconditions are satisfied as long as there exists at least one binding of the variable that satisfies the corresponding literal.

It is also possible to use the same predicates in the rule postconditions and preconditions, enabling the description of recursive rules. For example, the two rules at the beginning of this chapter provide a recursive definition of the concept $Ancestor(x, y)$. ILP learning methods such as those described below have been demonstrated to learn a variety of simple recursive functions, such as the above *Ancestor* function, and functions for sorting the elements of a list, removing a specific element from a list, and appending two lists.

10.4.2 Terminology

Before moving on to algorithms for learning sets of Horn clauses, let us introduce some basic terminology from formal logic. All expressions are composed of *constants* (e.g., *Bob*, *Louise*), *variables* (e.g., x, y), *predicate* symbols (e.g., *Married*, *Greater_Than*), and *function* symbols (e.g., *age*). The difference between predicates and functions is that predicates take on values of *True* or *False*, whereas functions may take on any constant as their value. We will use lowercase symbols for variables and capitalized symbols for constants. Also, we will use lowercase for functions and capitalized symbols for predicates.

From these symbols, we build up expressions as follows: A *term* is any constant, any variable, or any function applied to any term (e.g., *Bob*, x, *age*(*Bob*)). A *literal* is any predicate or its negation applied to any term (e.g., *Married*(*Bob*, *Louise*), $\neg$*Greater_Than*(*age*(*Sue*), 20)). If a literal contains a negation ($\neg$) symbol, we call it a *negative literal*, otherwise a *positive literal.*

A *clause* is any disjunction of literals, where all variables are assumed to be universally quantified. A *Horn clause* is a clause containing at most one positive literal, such as

$$H \vee \neg L_1 \vee \ldots \neg L_n$$

where H is the positive literal, and $\neg L_1 \ldots \neg L_n$ are negative literals. Because of the equalities $(B \vee \neg A) = (B \leftarrow A)$ and $\neg(A \wedge B) = (\neg A \vee \neg B)$, the above Horn clause can alternatively be written in the form

$$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

- Every well-formed expression is composed of *constants* (e.g., $Mary$, 23, or Joe), *variables* (e.g., x), *predicates* (e.g., $Female$, as in $Female(Mary)$), and *functions* (e.g., age, as in $age(Mary)$).
- A *term* is any constant, any variable, or any function applied to any term. Examples include $Mary$, x, $age(Mary)$, $age(x)$.
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include $Female(Mary)$, $\neg Female(x)$, $Greater\_than(age(Mary), 20)$.
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg Female(Joe)$).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg Female(Joe)$).
- A *positive literal* is a literal with no negation sign (e.g., $Female(Mary)$).
- A *clause* is any disjunction of literals $M_1 \vee \ldots M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

$$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

  where $H, L_1 \ldots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.
- For any literals A and B, the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

$$H \vee \neg L_1 \vee \ldots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z. Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L.
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.

TABLE 10.3
Basic definitions from first-order logic.

which is equivalent to the following, using our earlier rule notation

$$\text{IF } L_1 \wedge \ldots \wedge L_n, \text{ THEN } H$$

Whatever the notation, the Horn clause preconditions $L_1 \wedge \ldots \wedge L_n$ are called the clause *body* or, alternatively, the clause *antecedents*. The literal H that forms the postcondition is called the clause *head* or, alternatively, the clause *consequent*. For easy reference, these definitions are summarized in Table 10.3, along with other definitions introduced later in this chapter.

10.5 LEARNING SETS OF FIRST-ORDER RULES: FOIL

A variety of algorithms has been proposed for learning first-order rules, or Horn clauses. In this section we consider a program called FOIL (Quinlan 1990) that employs an approach very similar to the SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithms of the previous section. In fact, the FOIL program is the natural extension of these earlier algorithms to first-order representations. Formally, the hypotheses learned by FOIL are sets of first-order rules, where each rule is similar to a Horn clause with two exceptions. First, the rules learned by FOIL are

more restricted than general Horn clauses, because the literals are not permitted to contain function symbols (this reduces the complexity of the hypothesis space search). Second, FOIL rules are more expressive than Horn clauses, because the literals appearing in the body of the rule may be negated. FOIL has been applied to a variety of problem domains. For example, it has been demonstrated to learn a recursive definition of the QUICKSORT algorithm and to learn to discriminate legal from illegal chess positions.

The FOIL algorithm is summarized in Table 10.4. Notice the outer loop corresponds to a variant of the SEQUENTIAL-COVERING algorithm discussed earlier; that is, it learns new rules one at a time, removing the positive examples covered by the latest rule before attempting to learn the next rule. The inner loop corresponds to a variant of our earlier LEARN-ONE-RULE algorithm, extended to accommodate first-order rules. Note also there are a few minor differences between FOIL and these earlier algorithms. In particular, FOIL seeks only rules that predict when the target literal is *True*, whereas our earlier algorithm would seek both rules that predict when it is *True* and rules that predict when it is *False*. Also, FOIL performs a simple hillclimbing search rather than a beam search (equivalently, it uses a beam of width one).

The hypothesis space search performed by FOIL is best understood by viewing it hierarchically. Each iteration through FOIL's outer loop adds a new rule to its disjunctive hypothesis, *Learned_rules*. The effect of each new rule is to gen-

---

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* ← those *Examples* for which the *Target_predicate* is *True*
- *Neg* ← those *Examples* for which the *Target_predicate* is *False*
- *Learned_rules* ← {}
- while *Pos*, do
  Learn a NewRule
  - *NewRule* ← the rule that predicts *Target_predicate* with no preconditions
  - *NewRuleNeg* ← *Neg*
  - while *NewRuleNeg*, do
    Add a new literal to specialize NewRule
    - *Candidate_literals* ← generate candidate new literals for *NewRule*, based on *Predicates*
    - *Best_literal* ← $\underset{L \in Candidate\_literals}{\text{argmax}}$ *Foil_Gain*(*L*, *NewRule*)
    - add *Best_literal* to preconditions of *NewRule*
    - *NewRuleNeg* ← subset of *NewRuleNeg* that satisfies *NewRule* preconditions
  - *Learned_rules* ← *Learned_rules* + *NewRule*
  - *Pos* ← *Pos* − {members of *Pos* covered by *NewRule*}
- Return *Learned_rules*

TABLE 10.4
The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.

eralize the current disjunctive hypothesis (i.e., to increase the number of instances it classifies as positive), by adding a new disjunct. Viewed at this level, the search is a *specific-to-general* search through the space of hypotheses, beginning with the most specific empty disjunction and terminating when the hypothesis is sufficiently general to cover all positive training examples. The inner loop of FOIL performs a finer-grained search to determine the exact definition of each new rule. This inner loop searches a second hypothesis space, consisting of conjunctions of literals, to find a conjunction that will form the preconditions for the new rule. Within this hypothesis space, it conducts a *general-to-specific*, hill-climbing search, beginning with the most general preconditions possible (the empty precondition), then adding literals one at a time to specialize the rule until it avoids all negative examples.

The two most substantial differences between FOIL and our earlier SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithm follow from the requirement that it accommodate first-order rules. These differences are:

1. In its general-to-specific search to learn each new rule, FOIL employs different detailed steps to generate candidate specializations of the rule. This difference follows from the need to accommodate variables in the rule preconditions.
2. FOIL employs a PERFORMANCE measure, $Foil\_Gain$, that differs from the entropy measure shown for LEARN-ONE-RULE in Table 10.2. This difference follows from the need to distinguish between different bindings of the rule variables and from the fact that FOIL seeks only rules that cover positive examples.

The following two subsections consider these two differences in greater detail.

10.5.1 Generating Candidate Specializations in FOIL

To generate candidate specializations of the current rule, FOIL generates a variety of new literals, each of which may be individually added to the rule preconditions. More precisely, suppose the current rule being considered is

$$P(x_1, x_2, \ldots, x_k) \leftarrow L_1 \ldots L_n$$

where $L_1 \ldots L_n$ are literals forming the current rule preconditions and where $P(x_1, x_2, \ldots, x_k)$ is the literal that forms the rule head, or postconditions. FOIL generates candidate specializations of this rule by considering new literals L_{n+1} that fit one of the following forms:

- $Q(v_1, \ldots, v_r)$, where Q is any predicate name occurring in *Predicates* and where the v_i are either new variables or variables already present in the rule. At least one of the v_i in the created literal must already exist as a variable in the rule.
- $Equal(x_j, x_k)$, where x_j and x_k are variables already present in the rule.

- The negation of either of the above forms of literals.

To illustrate, consider learning rules to predict the target literal $GrandDaughter(x, y)$, where the other predicates used to describe examples are $Father$ and $Female$. The general-to-specific search in FOIL begins with the most general rule

$$GrandDaughter(x, y) \leftarrow$$

which asserts that $GrandDaughter(x, y)$ is true of any x and y. To specialize this initial rule, the above procedure generates the following literals as candidate additions to the rule preconditions: $Equal(x, y)$, $Female(x)$, $Female(y)$, $Father(x, y)$, $Father(y, x)$, $Father(x, z)$, $Father(z, x)$, $Father(y, z)$, $Father(z, y)$, and the negations of each of these literals (e.g., $\neg Equal(x, y)$). Note that z is a new variable here, whereas x and y exist already within the current rule.

Now suppose that among the above literals FOIL greedily selects $Father(y, z)$ as the most promising, leading to the more specific rule

$$GrandDaughter(x, y) \leftarrow Father(y, z)$$

In generating candidate literals to further specialize this rule, FOIL will now consider all of the literals mentioned in the previous step, plus the additional literals $Female(z)$, $Equal(z, x)$, $Equal(z, y)$, $Father(z, w)$, $Father(w, z)$, and their negations. These new literals are considered at this point because the variable z was added to the rule in the previous step. Because of this, FOIL now considers an additional new variable w.

If FOIL at this point were to select the literal $Father(z, x)$ and on the next iteration select the literal $Female(y)$, this would lead to the following rule, which covers only positive examples and hence terminates the search for further specializations of the rule.

$$GrandDaughter(x, y) \leftarrow Father(y, z) \wedge Father(z, x) \wedge Female(y)$$

At this point, FOIL will remove all positive examples covered by this new rule. If additional positive examples remain to be covered, then it will begin yet another general-to-specific search for an additional rule.

10.5.2 Guiding the Search in FOIL

To select the most promising literal from the candidates generated at each step, FOIL considers the performance of the rule over the training data. In doing this, it considers all possible bindings of each variable in the current rule. To illustrate this process, consider again the example in which we seek to learn a set of rules for the target literal $GrandDaughter(x, y)$. For illustration, assume the training data includes the following simple set of assertions, where we use the convention that $P(x, y)$ can be read as "The P of x is y."

$GrandDaughter(Victor, Sharon)$	$Father(Sharon, Bob)$	$Father(Tom, Bob)$
$Female(Sharon)$	$Father(Bob, Victor)$	

Here let us also make the closed world assumption that any literal involving the predicate *GrandDaughter*, *Father*, or *Female* and the constants *Victor*, *Sharon*, *Bob*, and *Tom* that is not listed above can be assumed to be false (i.e., we also implicitly assert $\neg GrandDaughter(Tom, Bob)$, $\neg GrandDaughter(Victor, Victor)$, etc.).

To select the best specialization of the current rule, FOIL considers each distinct way in which the rule variables can bind to constants in the training examples. For example, in the initial step when the rule is

$$GrandDaughter(x, y) \leftarrow$$

the rule variables x and y are not constrained by any preconditions and may therefore bind in any combination to the four constants *Victor*, *Sharon*, *Bob*, and *Tom*. We will use the notation $\{x/Bob, y/Sharon\}$ to denote a particular variable binding; that is, a substitution mapping each variable to a constant. Given the four possible constants, there are 16 possible variable bindings for this initial rule. The binding $\{x/Victor, y/Sharon\}$ corresponds to a positive example binding, because the training data includes the assertion $GrandDaughter(Victor, Sharon)$. The other 15 bindings allowed by the rule (e.g., the binding $\{x/Bob, y/Tom\}$) constitute negative evidence for the rule in the current example, because no corresponding assertion can be found in the training data.

At each stage, the rule is evaluated based on these sets of positive and negative variable bindings, with preference given to rules that possess more positive bindings and fewer negative bindings. As new literals are added to the rule, the sets of bindings will change. Note if a literal is added that introduces a new variable, then the bindings for the rule will grow in length (e.g., if $Father(y, z)$ is added to the above rule, then the original binding $\{x/Victor, y/Sharon\}$ will become the more lengthy $\{x/Victor, y/Sharon, z/Bob\}$. Note also that if the new variable can bind to several different constants, then the number of bindings fitting the extended rule can be greater than the number associated with the original rule.

The evaluation function used by FOIL to estimate the utility of adding a new literal is based on the numbers of positive and negative bindings covered before and after adding the new literal. More precisely, consider some rule R, and a candidate literal L that might be added to the body of R. Let R' be the rule created by adding literal L to rule R. The value $Foil\_Gain(L, R)$ of adding L to R is defined as

$$Foil\_Gain(L, R) \equiv t\left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0}\right) \tag{10.1}$$

where p_0 is the number of positive bindings of rule R, n_0 is the number of negative bindings of R, p_1 is the number of positive bindings of rule R', and n_1 is the number of negative bindings of R'. Finally, t is the number of positive bindings of rule R that are still covered after adding literal L to R. When a new variable is introduced into R by adding L, then any original binding is considered to be covered so long as some binding extending it is present in the bindings of R'.

This *Foil_Gain* function has a straightforward interpretation in terms of information theory. According to information theory, $-\log_2 \frac{p_0}{p_0+n_0}$ is the minimum number of bits needed to encode the classification of an arbitrary positive binding among the bindings covered by rule R. Similarly, $-\log_2 \frac{p_1}{p_1+n_1}$ is the number of bits required if the binding is one of those covered by rule R'. Since t is just the number of positive bindings covered by R that remain covered by R', $Foil\_Gain(L, R)$ can be seen as the reduction due to L in the total number of bits needed to encode the classification of all positive bindings of R.

10.5.3 Learning Recursive Rule Sets

In the above discussion, we ignored the possibility that new literals added to the rule body could refer to the target predicate itself (i.e., the predicate occurring in the rule head). However, if we include the target predicate in the input list of *Predicates*, then FOIL will consider it as well when generating candidate literals. This will allow it to form recursive rules—rules that use the same predicate in the body and the head of the rule. For instance, recall the following rule set that provides a recursive definition of the *Ancestor* relation.

IF $Parent(x, y)$ THEN $Ancestor(x, y)$
IF $Parent(x, z) \wedge Ancestor(z, y)$ THEN $Ancestor(x, y)$

Given an appropriate set of training examples, these two rules can be learned following a trace similar to the one above for *GrandDaughter*. Note the second rule is among the rules that are potentially within reach of FOIL's search, provided *Ancestor* is included in the list *Predicates* that determines which predicates may be considered when generating new literals. Of course whether this particular rule would be learned or not depends on whether these particular literals outscore competing candidates during FOIL's greedy search for increasingly specific rules. Cameron-Jones and Quinlan (1993) discuss several examples in which FOIL has successfully discovered recursive rule sets. They also discuss important subtleties that arise, such as how to avoid learning rule sets that produce infinite recursion.

10.5.4 Summary of FOIL

To summarize, FOIL extends the sequential covering algorithm of CN2 to handle the case of learning first-order rules similar to Horn clauses. To learn each rule FOIL performs a general-to-specific search, at each step adding a single new literal to the rule preconditions. The new literal may refer to variables already mentioned in the rule preconditions or postconditions, and may introduce new variables as well. At each step, it uses the *Foil_Gain* function of Equation (10.1) to select among the candidate new literals. If new literals are allowed to refer to the target predicate, then FOIL can, in principle, learn sets of recursive rules. While this introduces the complexity of avoiding rule sets that result in infinite recursion, FOIL has been demonstrated to successfully learn recursive rule sets in several cases.

In the case of noise-free training data, FOIL may continue adding new literals to the rule until it covers no negative examples. To handle noisy data, the search is continued until some tradeoff occurs between rule accuracy, coverage, and complexity. FOIL uses a minimum description length approach to halt the growth of rules, in which new literals are added only when their description length is shorter than the description length of the training data they explain. The details of this strategy are given in Quinlan (1990). In addition, FOIL post-prunes each rule it learns, using the same rule post-pruning strategy used for decision trees (Chapter 3).

10.6 INDUCTION AS INVERTED DEDUCTION

A second, quite different approach to inductive logic programming is based on the simple observation that induction is just the inverse of deduction! In general, machine learning involves building theories that explain the observed data. Given some data D and some partial background knowledge B, learning can be described as generating a hypothesis h that, together with B, explains D. Put more precisely, assume as usual that the training data D is a set of training examples, each of the form $\langle x_i, f(x_i)\rangle$. Here x_i denotes the ith training instance and $f(x_i)$ denotes its target value. Then learning is the problem of discovering a hypothesis h, such that the classification $f(x_i)$ of each training instance x_i follows deductively from the hypothesis h, the description of x_i, and any other background knowledge B known to the system.

$$(\forall \langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i) \tag{10.2}$$

The expression $X \vdash Y$ is read "Y follows deductively from X," or alternatively "X entails Y." Expression (10.2) describes the constraint that must be satisfied by the learned hypothesis h; namely, for every training instance x_i, the target classification $f(x_i)$ must follow deductively from B, h, and x_i.

As an example, consider the case where the target concept to be learned is "pairs of people $\langle u, v\rangle$ such that the child of u is v," represented by the predicate $Child(u, v)$. Assume we are given a single positive example $Child(Bob, Sharon)$, where the instance is described by the literals $Male(Bob)$, $Female(Sharon)$, and $Father(Sharon, Bob)$. Furthermore, suppose we have the general background knowledge $Parent(u, v) \leftarrow Father(u, v)$. We can describe this situation in the terms of Equation (10.2) as follows:

$$x_i : Male(Bob), Female(Sharon), Father(Sharon, Bob)$$

$$f(x_i) : Child(Bob, Sharon)$$

$$B : Parent(u, v) \leftarrow Father(u, v)$$

In this case, two of the many hypotheses that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$ are

$$h_1 : Child(u, v) \leftarrow Father(v, u)$$

$$h_2 : Child(u, v) \leftarrow Parent(v, u)$$

Note that the target literal $Child(Bob, Sharon)$ is entailed by $h_1 \wedge x_i$ with no need for the background information B. In the case of hypothesis h_2, however, the situation is different. The target $Child(Bob, Sharon)$ follows from $B \wedge h_2 \wedge x_i$, but not from $h_2 \wedge x_i$ alone. This example illustrates the role of background knowledge in expanding the set of acceptable hypotheses for a given set of training data. It also illustrates how new predicates (e.g., $Parent$) can be introduced into hypotheses (e.g., h_2), even when the predicate is not present in the original description of the instance x_i. This process of augmenting the set of predicates, based on background knowledge, is often referred to as *constructive induction.*

The significance of Equation (10.2) is that it casts the learning problem in the framework of deductive inference and formal logic. In the case of propositional and first-order logics, there exist well-understood algorithms for automated deduction. Interestingly, it is possible to develop inverses of these procedures in order to automate the process of inductive generalization. The insight that induction might be performed by inverting deduction appears to have been first observed by the nineteenth century economist W. S. Jevons, who wrote:

> Induction is, in fact, the inverse operation of deduction, and cannot be conceived to exist without the corresponding operation, so that the question of relative importance cannot arise. Who thinks of asking whether addition or subtraction is the more important process in arithmetic? But at the same time much difference in difficulty may exist between a direct and inverse operation; ... it must be allowed that inductive investigations are of a far higher degree of difficulty and complexity than any questions of deduction.... (Jevons 1874)

In the remainder of this chapter we will explore this view of induction as the inverse of deduction. The general issue we will be interested in here is designing *inverse entailment operators.* An inverse entailment operator, $O(B, D)$ takes the training data $D = \{\langle x_i, f(x_i)\rangle\}$ and background knowledge B as input and produces as output a hypothesis h satisfying Equation (10.2).

$$O(B, D) = h \text{ such that } (\forall \langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

Of course there will, in general, be many different hypotheses h that satisfy $(\forall \langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$. One common heuristic in ILP for choosing among such hypotheses is to rely on the heuristic known as the Minimum Description Length principle (see Section 6.6).

There are several attractive features to formulating the learning task as finding a hypothesis h that solves the relation $(\forall \langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$.

- This formulation subsumes the common definition of learning as finding some general concept that matches a given set of training examples (which corresponds to the special case where no background knowledge B is available).
- By incorporating the notion of background information B, this formulation allows a more rich definition of when a hypothesis may be said to "fit" the data. Up until now, we have always determined whether a hypothesis

(e.g., neural network) fits the data based solely on the description of the hypothesis and data, independent of the task domain under study. In contrast, this formulation allows the domain-specific background information B to become part of the definition of "fit." In particular, h fits the training example $\langle x_i, f(x_i)\rangle$ as long as $f(x_i)$ follows deductively from $B \wedge h \wedge x_i$.

- By incorporating background information B, this formulation invites learning methods that use this background information to guide the search for h, rather than merely searching the space of syntactically legal hypotheses. The inverse resolution procedure described in the following section uses background knowledge in this fashion.

At the same time, research on inductive logic programing following this formulation has encountered several practical difficulties.

- The requirement $(\forall\langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$ does not naturally accommodate noisy training data. The problem is that this expression does not allow for the possibility that there may be errors in the observed description of the instance x_i or its target value $f(x_i)$. Such errors can produce an inconsistent set of constraints on h. Unfortunately, most formal logic frameworks completely lose their ability to distinguish between truth and falsehood once they are given inconsistent sets of assertions.
- The language of first-order logic is so expressive, and the number of hypotheses that satisfy $(\forall\langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$ is so large, that the search through the space of hypotheses is intractable in the general case. Much recent work has sought restricted forms of first-order expressions, or additional second-order knowledge, to improve the tractability of the hypothesis space search.
- Despite our intuition that background knowledge B should help constrain the search for a hypothesis, in most ILP systems (including all discussed in this chapter) the complexity of the hypothesis space search *increases* as background knowledge B is increased. (However, see Chapters 11 and 12 for algorithms that use background knowledge to *decrease* rather than increase sample complexity).

In the following section, we examine one quite general inverse entailment operator that constructs hypotheses by inverting a deductive inference rule.

10.7 INVERTING RESOLUTION

A general method for automated deduction is the *resolution rule* introduced by Robinson (1965). The resolution rule is a sound and complete rule for deductive inference in first-order logic. Therefore, it is sensible to ask whether we can invert the resolution rule to form an inverse entailment operator. The answer is yes, and it is just this operator that forms the basis of the CIGOL program introduced by Muggleton and Buntine (1988).

It is easiest to introduce the resolution rule in propositional form, though it is readily extended to first-order representations. Let L be an arbitrary propositional literal, and let P and R be arbitrary propositional clauses. The resolution rule is

$$\begin{array}{lll} P & \vee & L \\ \neg L & \vee & R \\ \hline P & \vee & R \end{array}$$

which should be read as follows: Given the two clauses above the line, conclude the clause below the line. Intuitively, the resolution rule is quite sensible. Given the two assertions $P \vee L$ and $\neg L \vee R$, it is obvious that either L or $\neg L$ must be false. Therefore, either P or R must be true. Thus, the conclusion $P \vee R$ of the resolution rule is intuitively satisfying.

The general form of the propositional resolution operator is described in Table 10.5. Given two clauses C_1 and C_2, the resolution operator first identifies a literal L that occurs as a positive literal in one of these two clauses and as a negative literal in the other. It then draws the conclusion given by the above formula. For example, consider the application of the resolution operator illustrated on the left side of Figure 10.2. Given clauses C_1 and C_2, the first step of the procedure identifies the literal $L = \neg KnowMaterial$, which is present in C_1, and whose negation $\neg(\neg KnowMaterial) = KnowMaterial$ is present in C_2. Thus the conclusion is the clause formed by the union of the literals $C_1 - \{L\} = PassExam$ and $C_2 - \{\neg L\} = \neg Study$. As another example, the result of applying the resolution rule to the clauses $C_1 = A \vee B \vee C \vee \neg D$ and $C_2 = \neg B \vee E \vee F$ is the clause $A \vee C \vee \neg D \vee E \vee F$.

It is easy to invert the resolution operator to form an inverse entailment operator $O(C, C_1)$ that performs inductive inference. In general, the inverse entailment operator must derive one of the initial clauses, C_2, given the resolvent C and the other initial clause C_1. Consider an example in which we are given the resolvent $C = A \vee B$ and the initial clause $C_1 = B \vee D$. How can we derive a clause C_2 such that $C_1 \wedge C_2 \vdash C$? First, note that by the definition of the resolution operator, any literal that occurs in C but not in C_1 must have been present in C_2. In our example, this indicates that C_2 must contain the literal A. Second, the literal

1. Given initial clauses C_1 and C_2, find a literal L from clause C_1 such that $\neg L$ occurs in clause C_2.
2. Form the resolvent C by including all literals from C_1 and C_2, except for L and $\neg L$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

where $\cup$ denotes set union, and "$-$" denotes set difference.

TABLE 10.5
Resolution operator (propositional form). Given clauses C_1 and C_2, the resolution operator constructs a clause C such that $C_1 \wedge C_2 \vdash C$.

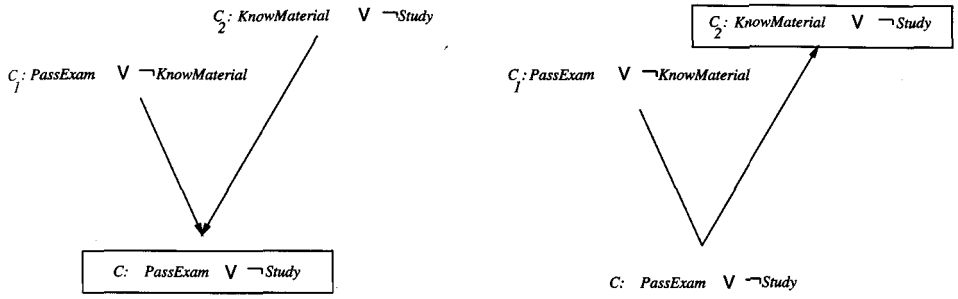

FIGURE 10.2
On the left, an application of the (deductive) resolution rule inferring clause C from the given clauses C_1 and C_2. On the right, an application of its (inductive) inverse, inferring C_2 from C and C_1.

that occurs in C_1 but not in C must be the literal removed by the resolution rule, and therefore its negation must occur in C_2. In our example, this indicates that C_2 must contain the literal $\neg D$. Hence, $C_2 = A \vee \neg D$. The reader can easily verify that applying the resolution rule to C_1 and C_2 does, in fact, produce the desired resolvent C.

Notice there is a second possible solution for C_2 in the above example. In particular, C_2 can also be the more specific clause $A \vee \neg D \vee B$. The difference between this and our first solution is that we have now included in C_2 a literal that occurred in C_1. The general point here is that inverse resolution is not deterministic—in general there may be multiple clauses C_2 such that C_1 and C_2 produce the resolvent C. One heuristic for choosing among the alternatives is to prefer shorter clauses over longer clauses, or equivalently, to assume C_2 shares no literals in common with C_1. If we incorporate this bias toward short clauses, the general statement of this inverse resolution procedure is as shown in Table 10.6.

We can develop rule-learning algorithms based on inverse entailment operators such as inverse resolution. In particular, the learning algorithm can use inverse entailment to construct hypotheses that, together with the background information, entail the training data. One strategy is to use a sequential covering algorithm to iteratively learn a set of Horn clauses in this way. On each iteration, the algorithm selects a training example $\langle x_i, f(x_i)\rangle$ that is not yet covered by previously learned clauses. The inverse resolution rule is then applied to

1. Given initial clauses C_1 and C, find a literal L that occurs in clause C_1, but not in clause C.
2. Form the second clause C_2 by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{\neg L\}$$

TABLE 10.6
Inverse resolution operator (propositional form). Given two clauses C and C_1, this computes a clause C_2 such that $C_1 \wedge C_2 \vdash C$.

generate candidate hypotheses h_i that satisfy $(B \wedge h_i \wedge x_i) \vdash f(x_i)$, where B is the background knowledge plus any clauses learned on previous iterations. Note this is an example-driven search, because each candidate hypothesis is constructed to cover a particular example. Of course if multiple candidate hypotheses exist, then one strategy for selecting among them is to choose the one with highest accuracy over the other examples as well. The CIGOL program uses inverse resolution with this kind of sequential covering algorithm, interacting with the user along the way to obtain training examples and to obtain guidance in its search through the vast space of possible inductive inference steps. However, CIGOL uses first-order rather than propositional representations. Below we describe the extension of the resolution rule required to accommodate first-order representations.

10.7.1 First-Order Resolution

The resolution rule extends easily to first-order expressions. As in the propositional case, it takes two clauses as input and produces a third clause as output. The key difference from the propositional case is that the process is now based on the notion of *unifying* substitutions.

We define a *substitution* to be any mapping of variables to terms. For example, the substitution $\theta = \{x/Bob, y/z\}$ indicates that the variable x is to be replaced by the term Bob, and that the variable y is to be replaced by the term z. We use the notation $W\theta$ to denote the result of applying the substitution θ to some expression W. For example, if L is the literal $Father(x, Bill)$ and θ is the substitution defined above, then $L\theta = Father(Bob, Bill)$.

We say that θ is a *unifying substitution* for two literals L_1 and L_2, provided $L_1\theta = L_2\theta$. For example, if $L_1 = Father(x, y)$, $L_2 = Father(Bill, z)$, and $\theta = \{x/Bill, z/y\}$, then θ is a unifying substitution for L_1 and L_2 because $L_1\theta = L_2\theta = Father(Bill, y)$. The significance of a unifying substitution is this: In the propositional form of resolution, the resolvent of two clauses C_1 and C_2 is found by identifying a literal L that appears in C_1 such that $\neg L$ appears in C_2. In first-order resolution, this generalizes to finding one literal L_1 from clause C_1 and one literal L_2 from C_2, such that some unifying substitution θ can be found for L_1 and $\neg L_2$ (i.e., such that $L_1\theta = \neg L_2\theta$). The resolution rule then constructs the resolvent C according to the equation

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta \tag{10.3}$$

The general statement of the resolution rule is shown in Table 10.7. To illustrate, suppose $C_1 = White(x) \leftarrow Swan(x)$ and suppose $C_2 = Swan(Fred)$. To apply the resolution rule, we first re-express C_1 in clause form as the equivalent expression $C_1 = White(x) \vee \neg Swan(x)$. The resolution rule can now be applied. In the first step, it finds the literal $L_1 = \neg Swan(x)$ from C_1 and the literal $L_2 = Swan(Fred)$ from C_2. If we choose the unifying substitution $\theta = \{x/Fred\}$ then these two literals satisfy $L_1\theta = \neg L_2\theta = \neg Swan(Fred)$. Therefore, the conclusion C is the union of $(C_1 - \{L_1\})\theta = White(Fred)$ and $(C_2 - \{L_2\})\theta = \emptyset$, or $C = White(Fred)$.

1. Find a literal L_1 from clause C_1, literal L_2 from clause C_2, and substitution θ such that $L_1\theta = \neg L_2\theta$.
2. Form the resolvent C by including all literals from $C_1\theta$ and $C_2\theta$, except for $L_1\theta$ and $\neg L_2\theta$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

TABLE 10.7
Resolution operator (first-order form).

10.7.2 Inverting Resolution: First-Order Case

We can derive the inverse resolution operator analytically, by algebraic manipulation of Equation (10.3) which defines the resolution rule. First, note the unifying substitution θ in Equation (10.3) can be uniquely factored into θ_1 and θ_2, where $\theta = \theta_1\theta_2$, where θ_1 contains all substitutions involving variables from clause C_1, and where θ_2 contains all substitutions involving variables from C_2. This factorization is possible because C_1 and C_2 will always begin with distinct variable names (because they are distinct universally quantified statements). Using this factorization of θ, we can restate Equation (10.3) as

$$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2$$

Keep in mind that "−" here stands for set difference. Now if we restrict inverse resolution to infer only clauses C_2 that contain no literals in common with C_1 (corresponding to a preference for shortest C_2 clauses), then we can re-express the above as

$$C - (C_1 - \{L_1\})\theta_1 = (C_2 - \{L_2\})\theta_2$$

Finally we use the fact that by definition of the resolution rule $L_2 = \neg L_1\theta_1\theta_2^{-1}$, and solve for C_2 to obtain

Inverse resolution:

$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1\theta_1\theta_2^{-1}\} \tag{10.4}$$

Equation (10.4) gives the inverse resolution rule for first-order logic. As in the propositional case, this inverse entailment operator is nondeterministic. In particular, in applying it we may in general find multiple choices for the clause C_1 to be resolved and for the unifying substitutions θ_1 and θ_2. Each set of choices may yield a different solution for C_2.

Figure 10.3 illustrates a multistep application of this inverse resolution rule for a simple example. In this figure, we wish to learn rules for the target predicate $GrandChild(y, x)$, given the training data $D = GrandChild(Bob, Shannon)$ and the background information $B = \{Father(Shannon, Tom), Father(Tom, Bob)\}$. Consider the bottommost step in the inverse resolution tree of Figure 10.3. Here, we set the conclusion C to the training example $GrandChild(Bob, Shannon)$

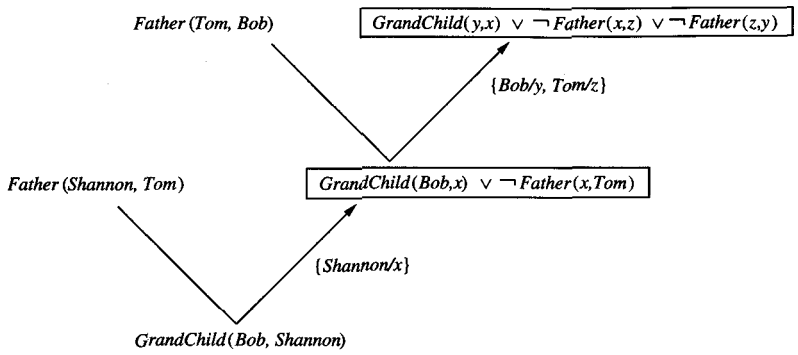

FIGURE 10.3
A multistep inverse resolution. In each case, the boxed clause is the result of the inference step. For each step, C is the clause at the bottom, C_1 the clause to the left, and C_2 the boxed clause to the right. In both inference steps here, θ_1 is the empty substitution {}, and θ_2^{-1} is the substitution shown below C_2. Note the final conclusion (the boxed clause at the top right) is the alternative form of the Horn clause $GrandChild(y, x) \leftarrow Father(x, z) \wedge Father(z, y)$.

and select the clause $C_1 = Father(Shannon, Tom)$ from the background information. To apply the inverse resolution operator we have only one choice for the literal L_1, namely $Father(Shannon, Tom)$. Suppose we choose the inverse substitutions $\theta_1^{-1} = \{\}$ and $\theta_2^{-1} = \{Shannon/x\}$. In this case, the resulting clause C_2 is the union of the clause $(C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} = (C\theta_1)\theta_2^{-1} = GrandChild(Bob, x)$, and the clause $\{\neg L_1\theta_1\theta_2^{-1}\} = \neg Father(x, Tom)$. Hence the result is the clause $GrandChild(Bob, x) \vee \neg Father(x, Tom)$, or equivalently $(GrandChild(Bob, x) \leftarrow Father(x, Tom))$. Note this general rule, together with C_1 entails the training example $GrandChild(Bob, Shannon)$.

In similar fashion, this inferred clause may now be used as the conclusion C for a second inverse resolution step, as illustrated in Figure 10.3. At each such step, note there are several possible outcomes, depending on the choices for the substitutions. (See Exercise 10.7.) In the example of Figure 10.3, the particular set of choices produces the intuitively satisfying final clause $GrandChild(y, x) \leftarrow Father(x, z) \wedge Father(z, y)$.

10.7.3 Summary of Inverse Resolution

To summarize, inverse resolution provides a general approach to automatically generating hypotheses h that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$. This is accomplished by inverting the general resolution rule given by Equation (10.3). Beginning with the resolution rule and solving for the clause C_2, the inverse resolution rule of Equation (10.4) is easily derived.

Given a set of beginning clauses, multiple hypotheses may be generated by repeated application of this inverse resolution rule. Note the inverse resolution rule has the advantage that it generates *only* hypotheses that satisfy $(B \wedge h \wedge x_i) \vdash f(x_i)$.

In contrast, the generate-and-test search of FOIL generates many hypotheses at each search step, including some that do not satisfy this constraint. FOIL then considers the data D to choose among these hypotheses. Given this difference, we might expect the search based on inverse resolution to be more focused and efficient. However, this will not necessarily be the case. One reason is that the inverse resolution operator can consider only a small fraction of the available data when generating its hypothesis at any given step, whereas FOIL considers all available data to select among its syntactically generated hypotheses. The differences between search strategies that use inverse entailment and those that use generate-and-test search is a subject of ongoing research. Srinivasan et al. (1995) provide one experimental comparison of these two approaches.

10.7.4 Generalization, θ-Subsumption, and Entailment

The previous section pointed out the correspondence between induction and inverse entailment. Given our earlier focus on using the general-to-specific ordering to organize the hypothesis search, it is interesting to consider the relationship between the *more_general_than* relation and inverse entailment. To illuminate this relationship, consider the following definitions.

- *more_general_than.* In Chapter 2, we defined the *more_general_than_or_equal_to* relation ($\geq_g$) as follows: Given two boolean-valued functions $h_j(x)$ and $h_k(x)$, we say that $h_j \geq_g h_k$ if and only if $(\forall x) h_k(x) \rightarrow h_j(x)$. This $\geq_g$ relation is used by many learning algorithms to guide search through the hypothesis space.
- *θ-subsumption.* Consider two clauses C_j and C_k, both of the form $H \vee L_1 \vee \ldots L_n$, where H is a positive literal, and the L_i are arbitrary literals. Clause C_j is said to *θ-subsume* clause C_k if and only if there exists a substitution θ such that $C_j\theta \subseteq C_k$ (where we here describe any clause C by the set of literals in its disjunctive form). This definition is due to Plotkin (1970).
- *Entailment.* Consider two clauses C_j and C_k. Clause C_j is said to *entail* clause C_k (written $C_j \vdash C_k$) if and only if C_k follows deductively from C_j.

What is the relationship among these three definitions? First, let us re-express the definition of $\geq_g$ using the same first-order notation as the other two definitions. If we consider a boolean-valued hypothesis $h(x)$ for some target concept $c(x)$, where $h(x)$ is expressed by a conjunction of literals, then we can re-express the hypothesis as the clause

$$c(x) \leftarrow h(x)$$

Here we follow the usual PROLOG interpretation that x is classified a negative example if it cannot be proven to be a positive example. Hence, we can see that our earlier definition of $\geq_g$ applies to the preconditions, or bodies, of Horn clauses. The implicit postcondition of the Horn clause is the target concept $c(x)$.

What is the relationship between this definition of $\geq_g$ and the definition of θ-subsumption? Note that if $h_1 \geq_g h_2$, then the clause $C_1 : c(x) \leftarrow h_1(x)$ θ-subsumes the clause $C_2 : c(x) \leftarrow h_2(x)$. Furthermore, θ-subsumption can hold even when the clauses have different heads. For example, clause A θ-subsumes clause B in the following case:

$$A: \quad Mother(x, y) \leftarrow Father(x, z) \wedge Spouse(z, y)$$

$$B: Mother(x, Louise) \leftarrow Father(x, Bob) \wedge Spouse(Bob, y) \wedge Female(x)$$

because $A\theta \subseteq B$ if we choose $\theta = \{y/Louise, z/Bob\}$. The key difference here is that $\geq_g$ implicitly assumes two clauses for which the heads are the same, whereas θ-subsumption can hold even for clauses with different heads.

Finally, θ-subsumption is a special case of entailment. That is, if clause A θ-subsumes clause B, then $A \vdash B$. However, we can find clauses A and B such that $A \vdash B$, but where A does not θ-subsume B. One example is the following pair of clauses

$$A: \quad Elephant(father\_of(x)) \leftarrow Elephant(x)$$

$$B: Elephant(father\_of(father\_of(y))) \leftarrow Elephant(y)$$

where $father\_of(x)$ is a function that refers to the individual who is the father of x. Note that although B can be proven from A, there is no substitution θ that allows B to be θ-subsumed by A.

As shown by these examples, our earlier notion of *more_general_than* is a special case of θ-subsumption, which is itself a special case of entailment. Therefore, searching the hypothesis space by generalizing or specializing hypotheses is more limited than searching by using general inverse entailment operators. Unfortunately, in its most general form, inverse entailment produces intractable searches. However, the intermediate notion of θ-subsumption provides one convenient notion that lies midway between our earlier definition of *more_general_than* and entailment.

10.7.5 PROGOL

Although inverse resolution is an intriguing method for generating candidate hypotheses, in practice it can easily lead to a combinatorial explosion of candidate hypotheses. An alternative approach is to use inverse entailment to generate just the single most specific hypothesis that, together with the background information, entails the observed data. This most specific hypothesis can then be used to bound a general-to-specific search through the hypothesis space similar to that used by FOIL, but with the additional constraint that the only hypotheses considered are hypotheses more general than this bound. This approach is employed by the PROGOL system, whose algorithm can be summarized as follows:

1. The user specifies a restricted language of first-order expressions to be used as the hypothesis space H. Restrictions are stated using "mode declarations,"

which enable the user to specify the predicate and function symbols to be considered, and the types and formats of arguments for each.

2. PROGOL uses a sequential covering algorithm to learn a set of expressions from H that cover the data. For each example $\langle x_i, f(x_i)\rangle$ that is not yet covered by these learned expressions, it first searches for the most specific hypothesis h_i within H such that $(B \wedge h_i \wedge x_i) \vdash f(x_i)$. More precisely, it approximates this by calculating the most specific hypothesis among those that entail $f(x_i)$ within k applications of the resolution rule (where k is a user-specified parameter).
3. PROGOL then performs a general-to-specific search of the hypothesis space bounded by the most general possible hypothesis and by the specific bound h_i calculated in step 2. Within this set of hypotheses, it seeks the hypothesis having minimum description length (measured by the number of literals). This part of the search is guided by an A^*-like heuristic that allows pruning without running the risk of pruning away the shortest hypothesis.

The details of the PROGOL algorithm are described by Muggleton (1992, 1995).

10.8 SUMMARY AND FURTHER READING

The main points of this chapter include:

- The sequential covering algorithm learns a disjunctive set of rules by first learning a single accurate rule, then removing the positive examples covered by this rule and iterating the process over the remaining training examples. It provides an efficient, greedy algorithm for learning rule sets, and an alternative to top-down decision tree learning algorithms such as ID3, which can be viewed as simultaneous, rather than sequential covering algorithms.
- In the context of sequential covering algorithms, a variety of methods have been explored for learning a single rule. These methods vary in the search strategy they use for examining the space of possible rule preconditions. One popular approach, exemplified by the CN2 program, is to conduct a general-to-specific beam search, generating and testing progressively more specific rules until a sufficiently accurate rule is found. Alternative approaches search from specific to general hypotheses, use an example-driven search rather than generate and test, and employ different statistical measures of rule accuracy to guide the search.
- Sets of first-order rules (i.e., rules containing variables) provide a highly expressive representation. For example, the programming language PROLOG represents general programs using collections of first-order Horn clauses. The problem of learning first-order Horn clauses is therefore often referred to as the problem of inductive logic programming.
- One approach to learning sets of first-order rules is to extend the sequential covering algorithm of CN2 from propositional to first-order representations.

This approach is exemplified by the FOIL program, which can learn sets of first-order rules, including simple recursive rule sets.

- A second approach to learning first-order rules is based on the observation that induction is the inverse of deduction. In other words, the problem of induction is to find a hypothesis h that satisfies the constraint

$$(\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

where B is general background information, $x_1 \ldots x_n$ are descriptions of the instances in the training data D, and $f(x_1) \ldots f(x_n)$ are the target values of the training instances.

- Following the view of induction as the inverse of deduction, some programs search for hypotheses by using operators that invert the well-known operators for deductive reasoning. For example, CIGOL uses inverse resolution, an operation that is the inverse of the deductive resolution operator commonly used for mechanical theorem proving. PROGOL combines an inverse entailment strategy with a general-to-specific strategy for searching the hypothesis space.

Early work on learning relational descriptions includes Winston's (1970) well-known program for learning network-style descriptions for concepts such as "arch." Banerji's (1964, 1969) work and Michalski's series of AQ programs (e.g., Michalski 1969; Michalski et al. 1986) were among the earliest to explore the use of logical representations in learning. Plotkin's (1970) definition of θ-subsumption provided an early formalization of the relationship between induction and deduction. Vere (1975) also explored learning logical representations, and Buchanan's (1976) META-DENDRAL program learned relational descriptions representing molecular substructures likely to fragment in a mass spectrometer. This program succeeded in discovering useful rules that were subsequently published in the chemistry literature. Mitchell's (1979) CANDIDATE-ELIMINATION version space algorithm was applied to these same relational descriptions of chemical structures.

With the popularity of the PROLOG language in the mid-1980s, researchers began to look more carefully at learning relational descriptions represented by Horn clauses. Early work on learning Horn clauses includes Shapiro's (1983) MIS and Sammut and Banerji's (1986) MARVIN. Quinlan's (1990) FOIL algorithm, discussed here, was quickly followed by a number of algorithms employing a general-to-specific search for first-order rules including MFOIL (Džeroski 1991), FOCL (Pazzani et al. 1991), CLAUDIEN (De Raedt and Bruynooghe 1993), and MARKUS (Grobelnik 1992). The FOCL algorithm is described in Chapter 12.

An alternative line of research on learning Horn clauses by inverse entailment was spurred by Muggleton and Buntine (1988), who built on related ideas by Sammut and Banerji (1986) and Muggleton (1987). More recent work along this line has focused on alternative search strategies and methods for constraining the hypothesis space to make learning more tractable. For example, Kietz and

Wrobel (1992) use rule schemata in their RDT program to restrict the form of expressions that may be considered during learning, and Muggleton and Feng (1992) discuss the restriction of first-order expressions to ij-determinate literals. Cohen (1994) discusses the GRENDEL program, which accepts as input an explicit description of the language for describing the clause body, thereby allowing the user to explicitly constrain the hypothesis space.

Lavrač and Džeroski (1994) provide a very readable textbook on inductive logic programming. Other useful recent monographs and edited collections include (Bergadano and Gunetti 1995; Morik et al. 1993; Muggleton 1992, 1995b). The overview chapter by Wrobel (1996) also provides a good perspective on the field. Bratko and Muggleton (1995) summarize a number of recent applications of ILP to problems of practical importance. A series of annual workshops on ILP provides a good source of recent research papers (e.g., see De Raedt 1996).

EXERCISES

10.1. Consider a sequential covering algorithm such as CN2 and a simultaneous covering algorithm such as ID3. Both algorithms are to be used to learn a target concept defined over instances represented by conjunctions of n boolean attributes. If ID3 learns a balanced decision tree of depth d, it will contain $2^d - 1$ distinct decision nodes, and therefore will have made $2^d - 1$ distinct choices while constructing its output hypothesis. How many rules will be formed if this tree is re-expressed as a disjunctive set of rules? How many preconditions will each rule possess? How many distinct choices would a *sequential* covering algorithm have to make to learn this same set of rules? Which system do you suspect would be more prone to overfitting if both were given the same training data?

10.2. Refine the LEARN-ONE-RULE algorithm of Table 10.2 so that it can learn rules whose preconditions include thresholds on real-valued attributes (e.g., *temperature* > 42). Specify your new algorithm as a set of editing changes to the algorithm of Table 10.2. Hint: Consider how this is accomplished for decision tree learning.

10.3. Refine the LEARN-ONE-RULE algorithm of Table 10.2 so that it can learn rules whose preconditions include constraints such as *nationality* $\in$ {*Canadian*, *Brazilian*}, where a discrete-valued attribute is allowed to take on any value in some specified set. Your modified program should explore the hypothesis space containing all such subsets. Specify your new algorithm as a set of editing changes to the algorithm of Table 10.2.

10.4. Consider the options for implementing LEARN-ONE-RULE in terms of the possible strategies for searching the hypothesis space. In particular, consider the following attributes of the search

(*a*) generate-and-test versus data-driven
(*b*) general-to-specific versus specific-to-general
(*c*) sequential cover versus simultaneous cover

Discuss the benefits of the choice made by the algorithm in Tables 10.1 and 10.2. For each of these three attributes of the search strategy, discuss the (positive and negative) impact of choosing the alternative option.

10.5. Apply inverse resolution in propositional form to the clauses $C = A \vee B$, $C_1 = A \vee B \vee G$. Give at least two possible results for C_2.

10.6. Apply inverse resolution to the clauses $C = R(B, x) \vee P(x, A)$ and $C_1 = S(B, y) \vee R(z, x)$. Give at least four possible results for C_2. Here A and B are constants, x and y are variables.

10.7. Consider the bottom-most inverse resolution step in Figure 10.3. Derive at least two different outcomes that could result given different choices for the substitutions θ_1 and θ_2. Derive a result for the inverse resolution step if the clause $Father(Tom, Bob)$ is used in place of $Father(Shannon, Tom)$.

10.8. Consider the relationship between the definition of the induction problem in this chapter

$$(\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

and our earlier definition of inductive bias from Chapter 2, Equation 2.1. There we defined the inductive bias, B_{bias}, by the expression

$$(\forall x_i \in X)(B_{bias} \wedge D \wedge x_i) \vdash L(x_i, D)$$

where $L(x_i, D)$ is the classification that the learner assigns to the new instance x_i after learning from the training data D, and where X is the entire instance space. Note the first expression is intended to describe the hypothesis we wish the learner to output, whereas the second expression is intended to describe the learner's policy for generalizing beyond the training data. Invent a learner for which the inductive bias B_{bias} of the learner is identical to the background knowledge B that it is provided.

REFERENCES

Banerji, R. (1964). A language for the description of concepts. *General Systems*, 9, 135–141.

Banerji, R. (1969). *Theory of problem solving—an approach to artificial intelligence*. New York: American Elsevier Publishing Company.

Bergadano, F., & Gunetti, D. (1995). *Inductive logic programming: From machine learning to software engineering*. Cambridge, Ma: MIT Press.

Bratko, I., & Muggleton, S. (1995). Applications of inductive logic programming. *Communications of the ACM*, 38(11), 65–70.

Buchanan, B. G., Smith, D. H., White, W. C., Gritter, R., Feigenbaum, E. A., Lederberg, J., & Djerassi, C. (1976). Applications of artificial intelligence for chemical inference, XXII: Automatic rule formation in mass spectrometry by means of the meta-DENDRAL program. *Journal of the American Chemical Society*, 98, 6168.

Buntine, W. (1986). Generalised subsumption. *Proceedings of the European Conference on Artificial Intelligence*, London.

Buntine, W. (1988). Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36, 149–176.

Cameron-Jones, R., & Quinlan, J. R. (1993). Avoiding pitfalls when learning recursive theories. *Proceedings of the Eighth International Workshop on Machine Learning* (pp 389–393). San Mateo, CA: Morgan Kaufmann.

Cestnik, B., & Bratko, I. (1991). On estimating probabilities in tree pruning. *Proceedings of the European Working Session on Machine Learning* (pp. 138–150). Porto, Portugal.

Clark, P., & Niblett, R. (1989). The CN2 induction algorithm. *Machine Learning*, 3, 261–284.

Cohen, W. (1994). Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68(2), 303–366.

De Raedt, L. (1992). *Interactive theory revision: An inductive logic programming approach*. London: Academic Press.

De Raedt, L., & Bruynooghe, M. (1993). A theory of clausal discovery. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann.

De Raedt, L. (Ed.). (1996). *Advances in inductive logic programming: Proceedings of the Fifth International Workshop on Inductive Logic Programming*. Amsterdam: IOS Press.

Dolsak, B., & Muggleton, S. (1992). The application of inductive logic programming to finite element mesh design. In S. Muggleton (Ed.), *Inductive Logic Programming*. London: Academic Press.

Džeroski, S. (1991). *Handling noise in inductive logic programming* (Master's thesis). Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia.

Flener, P. (1994). *Logic program synthesis from incomplete information.* The Kluwer international series in engineering and computer science. Boston: Kluwer Academic Publishers.

Grobelnik, M. (1992). MARKUS: An optimized model inference system. *Proceedings of the Workshop on Logical Approaches to Machine Learning, Tenth European Conference on AI*, Vienna, Austria.

Jevons, W. S. (1874). *The principles of science: A treatise on logic and scientific method.* London: Macmillan.

Kietz, J-U., & Wrobel, S. (1992). Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton (Ed.), *Inductive logic programming*. London: Academic Press.

Lavrač, N., & Džeroski, S. (1994). *Inductive logic programming: Techniques and applications*. Ellis Horwood.

Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., & Lederberg, J. (1980). *Applications of artificial intelligence for organic chemistry*. New York: McGraw-Hill.

Michalski, R. S., (1969). On the quasi-minimal solution of the general covering problem. *Proceedings of the First International Symposium on Information Processing* (pp. 125–128). Bled, Yugoslavia.

Michalski, R. S., Mozetic, I., Hong, J., and Lavrac, H. (1986). The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. *Proceedings of the Fifth National Conference on AI* (pp. 1041–1045). Philadelphia: Morgan-Kaufmann.

Mitchell, T. M. (1979). *Version spaces: An approach to concept learning* (Ph.D. dissertation). Electrical Engineering Dept., Stanford University, Stanford, CA.

Morik, K., Wrobel, S., Kietz, J.-U., & Emde, W. (1993). *Knowledge acquisition and machine learning: Theory, methods, and applications*. London: Academic Press.

Muggleton, S. (1987). DUCE: An oracle based approach to constructive induction. *Proceedings of the International Joint Conference on AI* (pp. 287–292). San Mateo, CA: Morgan Kaufmann.

Muggleton, S. (1995a). Inverse entailment and PROGOL. *New Generation Computing*, 13, 245–286.

Muggleton, S. (1995b). *Foundations of inductive logic programming*. Englewood Cliffs, NJ: Prentice Hall.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Machine Learning Conference* (pp. 339–352). Ann Arbor, Michigan: Morgan Kaufmann.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. *Proceedings of the First Conference on Algorithmic Learning Theory*. Ohmsha, Tokyo.

Muggleton, S., & Feng, C. (1992). Efficient induction of logic programs. In Muggleton (Ed.), *Inductive logic programming*. London: Academic Press.

Muggleton, S. (Ed.). (1992). *Inductive logic programming*. London: Academic Press.

Pazzani, M., Brunk, C., & Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 432–436). San Francisco: Morgan Kaufmann.

Plotkin, G. D. (1970). A note on inductive generalization. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5* (pp. 153–163). Edinburgh University Press.

Plotkin, G. D. (1971). A further note on inductive generalization. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 6*. New York: Elsevier.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, 5, 239–266.

Quinlan, J. R. (1991). *Improved estimates for the accuracy of small disjuncts* (Technical Note). *Machine Learning*, 6(1), 93–98. Boston: Kluwer Academic Publishers.

Rivest R. L. (1987). Learning decision lists. *Machine Learning*, 2(3), 229–246.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1), 23–41.

Sammut, C. A. (1981). Concept learning by experiment. *Seventh International Joint Conference on Artificial Intelligence*, Vancouver.

Sammut, C. A., & Banerji, R. B. (1986). Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol 2, pp. 167–192). Los Altos, California: Morgan Kaufmann.

Shapiro, E. (1983). *Algorithmic program debugging*. Cambridge MA: MIT Press.

Srinivasan, A., Muggleton, S., & King, R. D. (1995). *Comparing the use of background knowledge by inductive logic programming systems* (PRG Technical report PRG-TR-9-95). Oxford University Computing Laboratory.

Srinivasan, A., Muggleton, S., King, R. D., & Sternberg, M. J. E. (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. *Proceedings of the Fourth Inductive Logic Programming Workshop*.

Vere, S. (1975). Induction of concepts in the predicate calculus. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (pp. 351–356).

Winston, P. (1970). *Learning structural descriptions from examples* (Ph.D. dissertation) (MIT Technical Report AI-TR-231).

Wrobel, S. (1994). *Concept formation and knowledge revision*. Boston: Kluwer Academic Publishers.

Wrobel, S. (1996). Inductive logic programming. In G. Brewka (Ed)., *Principles of knowledge representation*. Stanford, CA: CSLI Publications.

CHAPTER 11

ANALYTICAL LEARNING

Inductive learning methods such as neural network and decision tree learning require a certain number of training examples to achieve a given level of generalization accuracy, as reflected in the theoretical bounds and experimental results discussed in earlier chapters. Analytical learning uses prior knowledge and deductive reasoning to augment the information provided by the training examples, so that it is not subject to these same bounds. This chapter considers an analytical learning method called explanation-based learning (EBL). In explanation-based learning, prior knowledge is used to analyze, or explain, how each observed training example satisfies the target concept. This explanation is then used to distinguish the relevant features of the training example from the irrelevant, so that examples can be generalized based on logical rather than statistical reasoning. Explanation-based learning has been successfully applied to learning search control rules for a variety of planning and scheduling tasks. This chapter considers explanation-based learning when the learner's prior knowledge is correct and complete. The next chapter considers combining inductive and analytical learning in situations where prior knowledge is only approximately correct.

11.1 INTRODUCTION

Previous chapters have considered a variety of *inductive* learning methods: that is, methods that generalize from observed training examples by identifying features that empirically distinguish positive from negative training examples. Decision tree learning, neural network learning, inductive logic programming, and genetic

algorithms are all examples of inductive methods that operate in this fashion. The key practical limit on these inductive learners is that they perform poorly when insufficient data is available. In fact, as discussed in Chapter 7, theoretical analysis shows that there are fundamental bounds on the accuracy that can be achieved when learning inductively from a given number of training examples.

Can we develop learning methods that are not subject to these fundamental bounds on learning accuracy imposed by the amount of training data available? Yes, if we are willing to reconsider the formulation of the learning problem itself. One way is to develop learning algorithms that accept explicit prior knowledge as an input, in addition to the input training data. Explanation-based learning is one such approach. It uses prior knowledge to analyze, or explain, each training example in order to infer which example features are relevant to the target function and which are irrelevant. These explanations enable it to generalize more accurately than inductive systems that rely on the data alone. As we saw in the previous chapter, inductive logic programming systems such as CIGOL also use prior background knowledge to guide learning. However, they use their background knowledge to infer features that augment the input descriptions of instances, thereby increasing the complexity of the hypothesis space to be searched. In contrast, explanation-based learning uses prior knowledge to *reduce* the complexity of the hypothesis space to be searched, thereby reducing sample complexity and improving generalization accuracy of the learner.

To capture the intuition underlying explanation-based learning, consider the task of learning to play chess. In particular, suppose we would like our chess program to learn to recognize important classes of game positions, such as the target concept "chessboard positions in which black will lose its queen within two moves." Figure 11.1 shows a positive training example of this target concept. Inductive learning methods could, of course, be employed to learn this target concept. However, because the chessboard is fairly complex (there are 32 pieces that may be on any of 64 squares), and because the particular patterns that capture this concept are fairly subtle (involving the relative positions of various pieces on the board), we would have to provide thousands of training examples similar to the one in Figure 11.1 to expect an inductively learned hypothesis to generalize correctly to new situations.

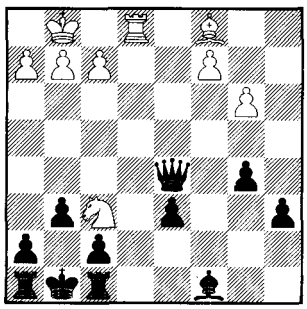

FIGURE 11.1
A positive example of the target concept "chess positions in which black will lose its queen within two moves." Note the white knight is simultaneously attacking both the black king and queen. Black must therefore move its king, enabling white to capture its queen.

What is interesting about this chess-learning task is that humans appear to learn such target concepts from just a handful of training examples! In fact, after considering only the single example shown in Figure 11.1, most people would be willing to suggest a general hypothesis for the target concept, such as "board positions in which the black king and queen are simultaneously attacked," and would not even consider the (equally consistent) hypothesis "board positions in which four white pawns are still in their original locations." How is it that humans can generalize so successfully from just this one example?

The answer appears to be that people rely heavily on explaining, or analyzing, the training example in terms of their prior knowledge about the legal moves of chess. If asked to explain why the training example of Figure 11.1 is a positive example of "positions in which the queen will be lost in two moves," most people would give an explanation similar to the following: "Because white's knight is attacking both the king and queen, black must move out of check, thereby allowing the knight to capture the queen." The importance of such explanations is that they provide the information needed to *rationally* generalize from the details of the training example to a correct general hypothesis. Features of the training example that are mentioned by the explanation (e.g., the position of the white knight, black king, and black queen) are relevant to the target concept and should be included in the general hypothesis. In contrast, features of the example that are not mentioned by the explanation (e.g., the fact that there are six black pawns on the board) can be assumed to be irrelevant details.

What exactly is the prior knowledge needed by a learner to construct the explanation in this chess example? It is simply knowledge about the legal rules of chess: knowledge of which moves are legal for the knight and other pieces, the fact that players must alternate moves in the game, and the fact that to win the game one player must capture his opponent's king. Note that given just this prior knowledge it is possible *in principle* to calculate the optimal chess move for any board position. However, in practice this calculation can be frustratingly complex and despite the fact that we humans ourselves possess this complete, perfect knowledge of chess, we remain unable to play the game optimally. As a result, much of human learning in chess (and in other search-intensive problems such as scheduling and planning) involves a long process of uncovering the consequences of our prior knowledge, guided by specific training examples encountered as we play the game.

This chapter describes learning algorithms that automatically construct and learn from such explanations. In the remainder of this section we define more precisely the analytical learning problem. The next section presents a particular explanation-based learning algorithm called PROLOG-EBG. Subsequent sections then examine the general properties of this algorithm and its relationship to inductive learning algorithms discussed in other chapters. The final section describes the application of explanation-based learning to improving performance at large state-space search problems. In this chapter we consider the special case in which explanations are generated from prior knowledge that is perfectly correct, as it is for us humans in the above chess example. In Chapter 12 we consider the more general case of learning when prior knowledge is only approximately correct.

11.1.1 Inductive and Analytical Learning Problems

The essential difference between analytical and inductive learning methods is that they assume two different formulations of the learning problem:

- In inductive learning, the learner is given a hypothesis space H from which it must select an output hypothesis, and a set of training examples $D = \{\langle x_1, f(x_1)\rangle, \ldots \langle x_n, f(x_n)\rangle\}$ where $f(x_i)$ is the target value for the instance x_i. The desired output of the learner is a hypothesis h from H that is consistent with these training examples.
- In analytical learning, the input to the learner includes the same hypothesis space H and training examples D as for inductive learning. In addition, the learner is provided an additional input: A *domain theory* B consisting of background knowledge that can be used to explain observed training examples. The desired output of the learner is a hypothesis h from H that is consistent with both the training examples D and the domain theory B.

To illustrate, in our chess example each instance x_i would describe a particular chess position, and $f(x_i)$ would be *True* when x_i is a position for which black will lose its queen within two moves, and *False* otherwise. We might define the hypothesis space H to consist of sets of Horn clauses (if-then rules) as in Chapter 10, where the predicates used by the rules refer to the positions or relative positions of specific pieces on the board. The domain theory B would consist of a formalization of the rules of chess, describing the legal moves, the fact that players must take turns, and the fact that the game is won when one player captures her opponent's king.

Note in analytical learning, the learner must output a hypothesis that is consistent with *both* the training data and the domain theory. We say that hypothesis h is *consistent* with domain theory B provided B does not entail the negation of h (i.e., $B \not\vdash \neg h$). This additional constraint that the output hypothesis must be consistent with B reduces the ambiguity faced by the learner when the data alone cannot resolve among all hypotheses in H. The net effect, provided the domain theory is correct, is to increase the accuracy of the output hypothesis.

Let us introduce in detail a second example of an analytical learning problem—one that we will use for illustration throughout this chapter. Consider an instance space X in which each instance is a pair of physical objects. Each of the two physical objects in the instance is described by the predicates *Color*, *Volume*, *Owner*, *Material*, *Type*, and *Density*, and the relationship between the two objects is described by the predicate *On*. Given this instance space, the task is to learn the target concept "pairs of physical objects, such that one can be stacked safely on the other," denoted by the predicate *SafeToStack(x,y)*. Learning this target concept might be useful, for example, to a robot system that has the task of storing various physical objects within a limited workspace. The full definition of this analytical learning task is given in Table 11.1.

Given:

- Instance space X: Each instance describes a pair of objects represented by the predicates $Type$, $Color$, $Volume$, $Owner$, $Material$, $Density$, and On.
- Hypothesis space H: Each hypothesis is a set of Horn clause rules. The head of each Horn clause is a literal containing the target predicate *SafeToStack*. The body of each Horn clause is a conjunction of literals based on the same predicates used to describe the instances, as well as the predicates $LessThan$, $Equal$, $GreaterThan$, and the functions $plus$, $minus$, and $times$. For example, the following Horn clause is in the hypothesis space:

$$SafeToStack(x, y) \leftarrow Volume(x, vx) \wedge Volume(y, vy) \wedge LessThan(vx, vy)$$

- Target concept: *SafeToStack(x,y)*
- Training Examples: A typical positive example, $SafeToStack(Obj1, Obj2)$, is shown below:

$On(Obj1, Obj2)$
$Type(Obj1, Box)$
$Type(Obj2, Endtable)$
$Color(Obj1, Red)$
$Color(Obj2, Blue)$
$Volume(Obj1, 2)$
$Owner(Obj1, Fred)$
$Owner(Obj2, Louise)$
$Density(Obj1, 0.3)$
$Material(Obj1, Cardboard)$
$Material(Obj2, Wood)$

- Domain Theory B:

$SafeToStack(x, y) \leftarrow \neg Fragile(y)$
$SafeToStack(x, y) \leftarrow Lighter(x, y)$
$Lighter(x, y) \leftarrow Weight(x, wx) \wedge Weight(y, wy) \wedge LessThan(wx, wy)$
$Weight(x, w) \leftarrow Volume(x, v) \wedge Density(x, d) \wedge Equal(w, times(v, d))$
$Weight(x, 5) \leftarrow Type(x, Endtable)$
$Fragile(x) \leftarrow Material(x, Glass)$
...

Determine:

- A hypothesis from H consistent with the training examples and domain theory.

TABLE 11.1
An analytical learning problem: *SafeToStack(x,y)*.

As shown in Table 11.1, we have chosen a hypothesis space H in which each hypothesis is a set of first-order if-then rules, or Horn clauses (throughout this chapter we follow the notation and terminology for first-order Horn clauses summarized in Table 10.3). For instance, the example Horn clause hypothesis shown in the table asserts that it is *SafeToStack* any object x on any object y, if the $Volume$ of x is $LessThan$ the $Volume$ of y (in this Horn clause the variables vx and vy represent the volumes of x and y, respectively). Note the Horn clause hypothesis can refer to any of the predicates used to describe the instances, as well as several additional predicates and functions. A typical positive training example, $SafeToStack(Obj1, Obj2)$, is also shown in the table.

To formulate this task as an analytical learning problem we must also provide a domain theory sufficient to explain why observed positive examples satisfy the target concept. In our earlier chess example, the domain theory corresponded to knowledge of the legal moves in chess, from which we constructed explanations

describing why black would lose its queen. In the current example, the domain theory must similarly explain why certain pairs of objects can be safely stacked on one another. The domain theory shown in the table includes assertions such as "it is safe to stack x on y if y is not *Fragile*," and "an object x is *Fragile* if the *Material* from which x is made is *Glass*." Like the learned hypothesis, the domain theory is described by a collection of Horn clauses, enabling the system in principle to incorporate any learned hypotheses into subsequent domain theories. Notice that the domain theory refers to additional predicates such as *Lighter* and *Fragile*, which are not present in the descriptions of the training examples, but which can be inferred from more primitive instance attributes such as *Material*, *Density*, and *Volume*, using other other rules in the domain theory. Finally, notice that the domain theory shown in the table is sufficient to prove that the positive example shown there satisfies the target concept *SafeToStack*.

11.2 LEARNING WITH PERFECT DOMAIN THEORIES: PROLOG-EBG

As stated earlier, in this chapter we consider explanation-based learning from domain theories that are perfect, that is, domain theories that are correct and complete. A domain theory is said to be *correct* if each of its assertions is a truthful statement about the world. A domain theory is said to be *complete* with respect to a given target concept and instance space, if the domain theory covers every positive example in the instance space. Put another way, it is complete if every instance that satisfies the target concept can be proven by the domain theory to satisfy it. Notice our definition of completeness does not require that the domain theory be able to prove that negative examples do not satisfy the target concept. However, if we follow the usual PROLOG convention that unprovable assertions are assumed to be false, then this definition of completeness includes full coverage of both positive and negative examples by the domain theory.

The reader may well ask at this point whether it is reasonable to assume that such perfect domain theories are available to the learner. After all, if the learner had a perfect domain theory, why would it need to learn? There are two responses to this question.

- First, there are cases in which it is feasible to provide a perfect domain theory. Our earlier chess problem provides one such case, in which the legal moves of chess form a perfect domain theory from which the optimal chess playing strategy can (in principle) be inferred. Furthermore, although it is quite easy to write down the legal moves of chess that constitute this domain theory, it is extremely difficult to write down the optimal chess-playing strategy. In such cases, we prefer to provide the domain theory to the learner and rely on the learner to formulate a useful description of the target concept (e.g., "board states in which I am about to lose my queen") by examining and generalizing from specific training examples. Section 11.4 describes the successful application of explanation-based learning with perfect domain

theories to automatically improve performance at several search-intensive planning and optimization problems.

- Second, in many other cases it is unreasonable to assume that a perfect domain theory is available. It is difficult to write a perfectly correct and complete theory even for our relatively simple *SafeToStack* problem. A more realistic assumption is that plausible explanations based on imperfect domain theories must be used, rather than exact proofs based on perfect knowledge. Nevertheless, we can begin to understand the role of explanations in learning by considering the ideal case of perfect domain theories. In Chapter 12 we will consider learning from imperfect domain theories.

This section presents an algorithm called PROLOG-EBG (Kedar-Cabelli and McCarty 1987) that is representative of several explanation-based learning algorithms. PROLOG-EBG is a sequential covering algorithm (see Chapter 10). In other words, it operates by learning a single Horn clause rule, removing the positive training examples covered by this rule, then iterating this process on the remaining positive examples until no further positive examples remain uncovered. When given a complete and correct domain theory, PROLOG-EBG is guaranteed to output a hypothesis (set of rules) that is itself correct and that covers the observed positive training examples. For any set of training examples, the hypothesis output by PROLOG-EBG constitutes a set of logically sufficient conditions for the target concept, according to the domain theory. PROLOG-EBG is a refinement of the EBG algorithm introduced by Mitchell et al. (1986) and is similar to the EGGS algorithm described by DeJong and Mooney (1986). The PROLOG-EBG algorithm is summarized in Table 11.2.

11.2.1 An Illustrative Trace

To illustrate, consider again the training example and domain theory shown in Table 11.1. As summarized in Table 11.2, the PROLOG-EBG algorithm is a sequential covering algorithm that considers the training data incrementally. For each new positive training example that is not yet covered by a learned Horn clause, it forms a new Horn clause by: (1) explaining the new positive training example, (2) analyzing this explanation to determine an appropriate generalization, and (3) refining the current hypothesis by adding a new Horn clause rule to cover this positive example, as well as other similar instances. Below we examine each of these three steps in turn.

11.2.1.1 EXPLAIN THE TRAINING EXAMPLE

The first step in processing each novel training example is to construct an explanation in terms of the domain theory, showing how this positive example satisfies the target concept. When the domain theory is correct and complete this explanation constitutes a *proof* that the training example satisfies the target concept. When dealing with imperfect prior knowledge, the notion of explanation must be extended to allow for plausible, approximate arguments rather than perfect proofs.

PROLOG-EBG(*TargetConcept*, *TrainingExamples*, *DomainTheory*)

- *LearnedRules* ← {}
- *Pos* ← the positive examples from *TrainingExamples*
- for each *PositiveExample* in *Pos* that is not covered by *LearnedRules*, do
  1. *Explain:*
     - *Explanation* ← an explanation (proof) in terms of the *DomainTheory* that *PositiveExample* satisfies the *TargetConcept*
  2. *Analyze:*
     - *SufficientConditions* ← the most general set of features of *PositiveExample* sufficient to satisfy the *TargetConcept* according to the *Explanation*.
  3. *Refine:*
     - *LearnedRules* ← *LearnedRules* + *NewHornClause*, where *NewHornClause* is of the form

$$TargetConcept \leftarrow SufficientConditions$$

- Return *LearnedRules*

TABLE 11.2
The explanation-based learning algorithm PROLOG-EBG. For each positive example that is not yet covered by the set of learned Horn clauses (*LearnedRules*), a new Horn clause is created. This new Horn clause is created by (1) explaining the training example in terms of the domain theory, (2) analyzing this explanation to determine the relevant features of the example, then (3) constructing a new Horn clause that concludes the target concept when this set of features is satisfied.

The explanation for the current training example is shown in Figure 11.2. Note the bottom of this figure depicts in graphical form the positive training example *SafeToStack*(*Obj1*, *Obj2*) from Table 11.1. The top of the figure depicts the explanation constructed for this training example. Notice the explanation, or proof, states that it is *SafeToStack Obj1* on *Obj2* because *Obj1* is *Lighter* than *Obj2*. Furthermore, *Obj1* is known to be *Lighter*, because its *Weight* can be inferred from its *Density* and *Volume*, and because the *Weight* of *Obj2* can be inferred from the default weight of an *Endtable*. The specific Horn clauses that underlie this explanation are shown in the domain theory of Table 11.1. Notice that the explanation mentions only a small fraction of the known attributes of *Obj1* and *Obj2* (i.e., those attributes corresponding to the shaded region in the figure).

While only a single explanation is possible for the training example and domain theory shown here, in general there may be multiple possible explanations. In such cases, any or all of the explanations may be used. While each may give rise to a somewhat different generalization of the training example, all will be justified by the given domain theory. In the case of PROLOG-EBG, the explanation is generated using a backward chaining search as performed by PROLOG. PROLOG-EBG, like PROLOG, halts once it finds the first valid proof.

11.2.1.2 ANALYZE THE EXPLANATION

The key question faced in generalizing the training example is "of the many features that happen to be true of the current training example, which ones are gen-

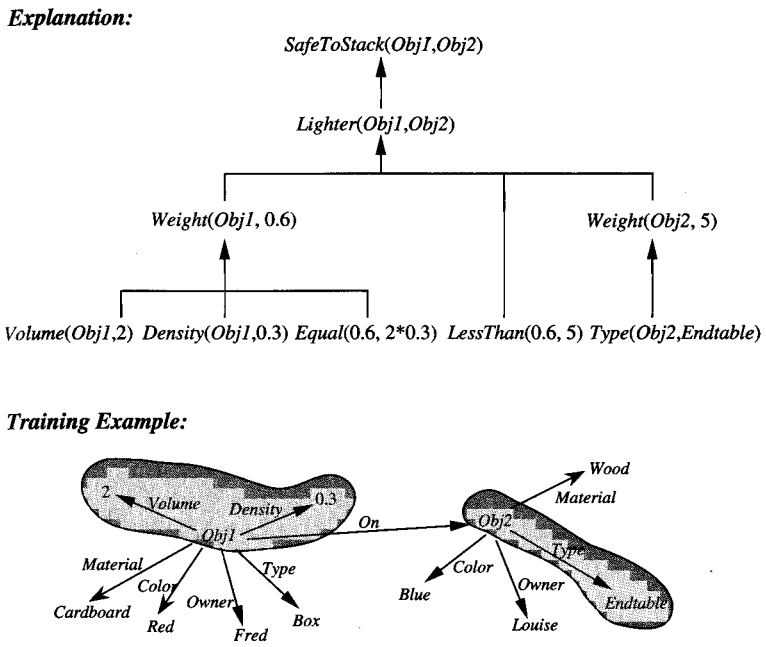

FIGURE 11.2
Explanation of a training example. The network at the bottom depicts graphically the training example *SafeToStack*(*Obj1*, *Obj2*) described in Table 11.1. The top portion of the figure depicts the explanation of how this example satisfies the target concept, *SafeToStack*. The shaded region of the training example indicates the example attributes used in the explanation. The other, irrelevant, example attributes will be dropped from the generalized hypothesis formed from this analysis.

erally relevant to the target concept?" The explanation constructed by the learner provides a direct answer to this question: precisely those features mentioned in the explanation. For example, the explanation of Figure 11.2 refers to the *Density* of *Obj1*, but not to its *Owner*. Therefore, the hypothesis for *SafeToStack(x,y)* should include $Density(x, 0.3)$, but not $Owner(x, Fred)$. By collecting just the features mentioned in the leaf nodes of the explanation in Figure 11.2 and substituting variables x and y for *Obj1* and *Obj2*, we can form a general rule that is justified by the domain theory:

$$SafeToStack(x, y) \leftarrow Volume(x, 2) \wedge Density(x, 0.3) \wedge Type(y, Endtable)$$

The body of the above rule includes each leaf node in the proof tree, except for the leaf nodes "$Equal(0.6, times(2, 0.3)$" and "$LessThan(0.6, 5)$." We omit these two because they are by definition always satisfied, independent of x and y.

Along with this learned rule, the program can also provide its justification: The explanation of the training example forms a proof for the correctness of this rule. Although this explanation was formed to cover the observed training example, the same explanation will apply to any instance that matches this general rule.

The above rule constitutes a significant generalization of the training example, because it omits many properties of the example (e.g., the *Color* of the two objects) that are irrelevant to the target concept. However, an even more general rule can be obtained by more careful analysis of the explanation. PROLOG-EBG computes the most general rule that can be justified by the explanation, by computing the *weakest preimage* of the explanation, defined as follows:

> ***Definition:*** The **weakest preimage** of a conclusion C with respect to a proof P is the most general set of initial assertions A, such that A entails C according to P.

For example, the weakest preimage of the target concept *SafeToStack(x,y)*, with respect to the explanation from Table 11.1, is given by the body of the following rule. This is the most general rule that can be justified by the explanation of Figure 11.2:

$$SafeToStack(x, y) \leftarrow Volume(x, vx) \wedge Density(x, dx) \wedge Equal(wx, times(vx, dx)) \wedge LessThan(wx, 5) \wedge Type(y, Endtable)$$

Notice this more general rule does not require the specific values for *Volume* and *Density* that were required by the first rule. Instead, it states a more general constraint on the values of these attributes.

PROLOG-EBG computes the weakest preimage of the target concept with respect to the explanation, using a general procedure called *regression* (Waldinger 1977). The regression procedure operates on a domain theory represented by an arbitrary set of Horn clauses. It works iteratively backward through the explanation, first computing the weakest preimage of the target concept with respect to the final proof step in the explanation, then computing the weakest preimage of the resulting expressions with respect to the preceding step, and so on. The procedure terminates when it has iterated over all steps in the explanation, yielding the weakest precondition of the target concept with respect to the literals at the leaf nodes of the explanation.

A trace of this regression process is illustrated in Figure 11.3. In this figure, the explanation from Figure 11.2 is redrawn in standard (nonitalic) font. The frontier of regressed expressions created at each step by the regression procedure is shown underlined in italics. The process begins at the root of the tree, with the frontier initialized to the general target concept *SafeToStack(x,y)*. The first step is to compute the weakest preimage of this frontier expression with respect to the final (top-most) inference rule in the explanation. The rule in this case is $SafeToStack(x, y) \leftarrow Lighter(x, y)$, so the resulting weakest preimage is $Lighter(x, y)$. The process now continues by regressing the new frontier, $\{Lighter(x, y)\}$, through the next Horn clause in the explanation, resulting in the regressed expressions $\{Weight(x, wx), LessThan(wx, wy), Weight(y, wy)\}$. This indicates that the explanation will hold for any x and y such that the weight wx of x is less than the weight wy of y. The regression of this frontier back to the leaf nodes of the explanation continues in this step-by-step fashion, finally

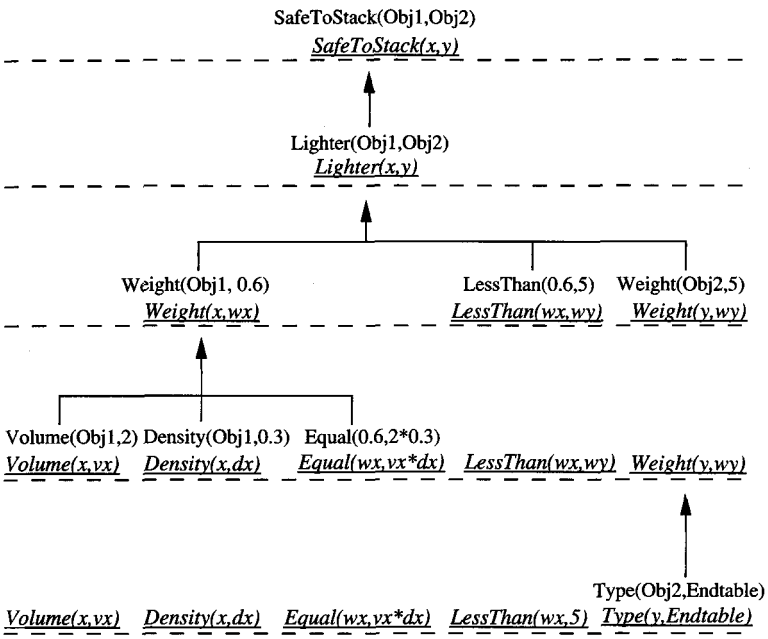

FIGURE 11.3
Computing the weakest preimage of *SafeToStack*($Obj1$, $Obj2$) with respect to the explanation. The target concept is regressed from the root (conclusion) of the explanation, down to the leaves. At each step (indicated by the dashed lines) the current frontier set of literals (underlined in italics) is regressed backward over one rule in the explanation. When this process is completed, the conjunction of resulting literals constitutes the weakest preimage of the target concept with respect to the explanation. This weakest preimage is shown by the italicized literals at the bottom of the figure.

resulting in a set of generalized literals for the leaf nodes of the tree. This final set of literals, shown at the bottom of Figure 11.3, forms the body of the final rule.

The heart of the regression procedure is the algorithm that at each step regresses the current frontier of expressions through a single Horn clause from the domain theory. This algorithm is described and illustrated in Table 11.3. The illustrated example in this table corresponds to the bottommost single regression step of Figure 11.3. As shown in the table, the REGRESS algorithm operates by finding a substitution that unifies the head of the Horn clause rule with the corresponding literal in the frontier, replacing this expression in the frontier by the rule body, then applying a unifying substitution to the entire frontier.

The final Horn clause rule output by PROLOG-EBG is formulated as follows: The clause body is defined to be the weakest preconditions calculated by the above procedure. The clause head is the target concept itself, with each substitution from each regression step (i.e., the substitution θ_{hl} in Table 11.3) applied to it. This substitution is necessary in order to keep consistent variable names between the head and body of the created clause, and to specialize the clause head when the

REGRESS(*Frontier, Rule, Literal,* θ_{hi})

Frontier: Set of literals to be regressed through Rule

Rule: A Horn clause

Literal: A literal in Frontier *that is inferred by Rule in the explanation*

θ_{hi}*: The substitution that unifies the head of Rule to the corresponding literal in the explanation*

Returns the set of literals forming the weakest preimage of Frontier with respect to Rule

- *head* ← *head* of *Rule*
- *body* ← *body* of *Rule*
- θ_{hl} ← the most general unifier of *head* with *Literal* such that there exists a substitution θ_{li} for which

$$\theta_{li}(\theta_{hl}(head)) = \theta_{hi}(head)$$

- Return $\theta_{hl}(Frontier - head + body)$

Example (the bottommost regression step in Figure 11.3):

REGRESS(*Frontier, Rule, Literal,* θ_{hi}) where

Frontier = $\{Volume(x, vs), Density(x, dx), Equal(wx, times(vx, dx)), LessThan(wx, wy), Weight(y, wy)\}$

Rule = $Weight(z, 5) \leftarrow Type(z, Endtable)$

Literal = $Weight(y, wy)$

$\theta_{hi} = \{z/Obj2\}$

- *head* ← $Weight(z, 5)$
- *body* ← $Type(z, Endtable)$
- θ_{hl} ← $\{z/y, wy/5\}$, where $\theta_{li} = \{y/Obj2\}$
- Return $\{Volume(x, vs), Density(x, dx), Equal(wx, times(vx, dx)), LessThan(wx, 5), Type(y, Endtable)\}$

TABLE 11.3
Algorithm for regressing a set of literals through a single Horn clause. The set of literals given by *Frontier* is regressed through *Rule*. *Literal* is the member of *Frontier* inferred by *Rule* in the explanation. The substitution θ_{hi} gives the binding of variables from the head of *Rule* to the corresponding literal in the explanation. The algorithm first computes a substitution θ_{hl} that unifies the *Rule* head to *Literal*, in a way that is consistent with the substitution θ_{hi}. It then applies this substitution θ_{hl} to construct the preimage of *Frontier* with respect to *Rule*. The symbols "+" and "–" in the algorithm denote set union and set difference. The notation $\{z/y\}$ denotes the substitution of y in place of z. An example trace is given.

explanation applies to only a special case of the target concept. As noted earlier, for the current example the final rule is

$$SafeToStack(x, y) \leftarrow Volume(x, vx) \wedge Density(x, dx) \wedge Equal(wx, times(vx, dx)) \wedge LessThan(wx, 5) \wedge Type(y, Endtable)$$

11.2.1.3 REFINE THE CURRENT HYPOTHESIS

The current hypothesis at each stage consists of the set of Horn clauses learned thus far. At each stage, the sequential covering algorithm picks a new positive

example that is not yet covered by the current Horn clauses, explains this new example, and formulates a new rule according to the procedure described above. Notice only positive examples are covered in the algorithm as we have defined it, and the learned set of Horn clause rules predicts only positive examples. A new instance is classified as negative if the current rules fail to predict that it is positive. This is in keeping with the standard negation-as-failure approach used in Horn clause inference systems such as PROLOG.

11.3 REMARKS ON EXPLANATION-BASED LEARNING

As we saw in the above example, PROLOG-EBG conducts a detailed analysis of individual training examples to determine how best to generalize from the specific example to a general Horn clause hypothesis. The following are the key properties of this algorithm.

- Unlike inductive methods, PROLOG-EBG produces *justified* general hypotheses by using prior knowledge to analyze individual examples.
- The explanation of how the example satisfies the target concept determines which example attributes are relevant: those mentioned by the explanation.
- The further analysis of the explanation, regressing the target concept to determine its weakest preimage with respect to the explanation, allows deriving more general constraints on the values of the relevant features.
- Each learned Horn clause corresponds to a sufficient condition for satisfying the target concept. The set of learned Horn clauses covers the positive training examples encountered by the learner, as well as other instances that share the same explanations.
- The generality of the learned Horn clauses will depend on the formulation of the domain theory and on the sequence in which training examples are considered.
- PROLOG-EBG implicitly assumes that the domain theory is correct and complete. If the domain theory is incorrect or incomplete, the resulting learned concept may also be incorrect.

There are several related perspectives on explanation-based learning that help to understand its capabilities and limitations.

- *EBL as theory-guided generalization of examples.* EBL uses its given domain theory to generalize *rationally* from examples, distinguishing the relevant example attributes from the irrelevant, thereby allowing it to avoid the bounds on sample complexity that apply to purely inductive learning. This is the perspective implicit in the above description of the PROLOG-EBG algorithm.
- *EBL as example-guided reformulation of theories.* The PROLOG-EBG algorithm can be viewed as a method for reformulating the domain theory into a more operational form. In particular, the original domain theory is reformulated by creating rules that (a) follow deductively from the domain theory,

and (b) classify the observed training examples in a single inference step. Thus, the learned rules can be seen as a reformulation of the domain theory into a set of special-case rules capable of classifying instances of the target concept in a single inference step.

- *EBL as "just" restating what the learner already "knows."* In one sense, the learner in our *SafeToStack* example begins with full knowledge of the *SafeToStack* concept. That is, if its initial domain theory is sufficient to explain any observed training examples, then it is also sufficient to predict their classification in advance. In what sense, then, does this qualify as learning? One answer is that in many tasks the difference between what one knows *in principle* and what one can efficiently compute *in practice* may be great, and in such cases this kind of "knowledge reformulation" can be an important form of learning. In playing chess, for example, the rules of the game constitute a perfect domain theory, sufficient in principle to play perfect chess. Despite this fact, people still require considerable experience to learn how to play chess well. This is precisely a situation in which a complete, perfect domain theory is already known to the (human) learner, and further learning is "simply" a matter of reformulating this knowledge into a form in which it can be used more effectively to select appropriate moves. A beginning course in Newtonian physics exhibits the same property—the basic laws of physics are easily stated, but students nevertheless spend a large part of a semester working out the consequences so they have this knowledge in more operational form and need not derive every problem solution from first principles come the final exam. PROLOG-EBG performs this type of reformulation of knowledge—its learned rules map directly from observable instance features to the classification relative to the target concept, in a way that is consistent with the underlying domain theory. Whereas it may require many inference steps and considerable search to classify an arbitrary instance using the original domain theory, the learned rules classify the observed instances in a single inference step.

Thus, in its pure form EBL involves reformulating the domain theory to produce general rules that classify examples in a single inference step. This kind of knowledge reformulation is sometimes referred to as *knowledge compilation*, indicating that the transformation is an efficiency improving one that does not alter the correctness of the system's knowledge.

11.3.1 Discovering New Features

One interesting capability of PROLOG-EBG is its ability to formulate new features that are not explicit in the description of the training examples, but that are needed to describe the general rule underlying the training example. This capability is illustrated by the algorithm trace and the learned rule in the previous section. In particular, the learned rule asserts that the essential constraint on the $Volume$ and $Density$ of x is that their product is less than 5. In fact, the training examples

contain no description of such a product, or of the value it should take on. Instead, this constraint is formulated automatically by the learner.

Notice this learned "feature" is similar in kind to the types of features represented by the hidden units of neural networks; that is, this feature is one of a very large set of potential features that can be computed from the available instance attributes. Like the BACKPROPAGATION algorithm, PROLOG-EBG automatically formulates such features in its attempt to fit the training data. However, unlike the statistical process that derives hidden unit features in neural networks from many training examples, PROLOG-EBG employs an analytical process to derive new features based on analysis of single training examples. Above, PROLOG-EBG derives the feature $Volume \cdot Density > 5$ analytically from the particular instantiation of the domain theory used to explain a single training example. For example, the notion that the product of *Volume* and *Density* is important arises from the domain theory rule that defines *Weight*. The notion that this product should be less than 5 arises from two other domain theory rules that assert that *Obj*1 should be *Lighter* than the *Endtable*, and that the *Weight* of the *Endtable* is 5. Thus, it is the particular composition and instantiation of these primitive terms from the domain theory that gives rise to defining this new feature.

The issue of automatically learning useful features to augment the instance representation is an important issue for machine learning. The analytical derivation of new features in explanation-based learning and the inductive derivation of new features in the hidden layer of neural networks provide two distinct approaches. Because they rely on different sources of information (statistical regularities over many examples versus analysis of single examples using the domain theory), it may be useful to explore new methods that combine both sources.

11.3.2 Deductive Learning

In its pure form, PROLOG-EBG is a deductive, rather than inductive, learning process. That is, by calculating the weakest preimage of the explanation it produces a hypothesis h that follows deductively from the domain theory B, while covering the training data D. To be more precise, PROLOG-EBG outputs a hypothesis h that satisfies the following two constraints:

$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (h \wedge x_i) \vdash f(x_i) \tag{11.1}$$

$$D \wedge B \vdash h \tag{11.2}$$

where the training data D consists of a set of training examples in which x_i is the ith training instance and $f(x_i)$ is its target value (f is the target function). Notice the first of these constraints is simply a formalization of the usual requirement in machine learning, that the hypothesis h correctly predict the target value $f(x_i)$ for each instance x_i in the training data.† Of course there will, in general, be many

†Here we include PROLOG-style negation-by-failure in our definition of entailment ($\vdash$), so that examples are entailed to be negative examples if they cannot be proven to be positive.

alternative hypotheses that satisfy this first constraint. The second constraint describes the impact of the domain theory in PROLOG-EBL: The output hypothesis is further constrained so that it must follow from the domain theory and the data. This second constraint reduces the ambiguity faced by the learner when it must choose a hypothesis. Thus, the impact of the domain theory is to reduce the effective size of the hypothesis space and hence reduce the sample complexity of learning.

Using similar notation, we can state the type of knowledge that is required by PROLOG-EBG for its domain theory. In particular, PROLOG-EBG assumes the domain theory B entails the classifications of the instances in the training data:

$$(\forall \langle x_i, f(x_i)\rangle \in D) \quad (B \wedge x_i) \vdash f(x_i) \tag{11.3}$$

This constraint on the domain theory B assures that an explanation can be constructed for each positive example.

It is interesting to compare the PROLOG-EBG learning setting to the setting for inductive logic programming (ILP) discussed in Chapter 10. In that chapter, we discussed a generalization of the usual inductive learning task, in which background knowledge B' is provided to the learner. We will use B' rather than B to denote the background knowledge used by ILP, because it does not typically satisfy the constraint given by Equation (11.3). ILP is an inductive learning system, whereas PROLOG-EBG is deductive. ILP uses its background knowledge B' to enlarge the set of hypotheses to be considered, whereas PROLOG-EBG uses its domain theory B to reduce the set of acceptable hypotheses. As stated in Equation (10.2), ILP systems output a hypothesis h that satisfies the following constraint:

$$(\forall \langle x_i, f(x_i)\rangle \in D) \quad (B' \wedge h \wedge x_i) \vdash f(x_i)$$

Note the relationship between this expression and the constraints on h imposed by PROLOG-EBG (given by Equations (11.1) and (11.2)). This ILP constraint on h is a weakened form of the constraint given by Equation (11.1)—the ILP constraint requires only that $(B' \wedge h \wedge x_i) \vdash f(x_i)$, whereas the PROLOG-EBG constraint requires the more strict $(h \wedge x_i) \vdash f(x_i)$. Note also that ILP imposes no constraint corresponding to the PROLOG-EBG constraint of Equation (11.2).

11.3.3 Inductive Bias in Explanation-Based Learning

Recall from Chapter 2 that the inductive bias of a learning algorithm is a set of assertions that, together with the training examples, deductively entail subsequent predictions made by the learner. The importance of inductive bias is that it characterizes how the learner generalizes beyond the observed training examples.

What is the inductive bias of PROLOG-EBG? In PROLOG-EBG the output hypothesis h follows deductively from $D \wedge B$, as described by Equation (11.2). Therefore, the domain theory B is a set of assertions which, together with the training examples, entail the output hypothesis. Given that predictions of the learner follow from this hypothesis h, it appears that the inductive bias of PROLOG-EBG is simply the domain theory B input to the learner. In fact, this is the case except for one

additional detail that must be considered: There are many alternative sets of Horn clauses entailed by the domain theory. The remaining component of the inductive bias is therefore the basis by which PROLOG-EBG chooses among these alternative sets of Horn clauses. As we saw above, PROLOG-EBG employs a sequential covering algorithm that continues to formulate additional Horn clauses until all positive training examples have been covered. Furthermore, each individual Horn clause is the most general clause (weakest preimage) licensed by the explanation of the current training example. Therefore, among the sets of Horn clauses entailed by the domain theory, we can characterize the bias of PROLOG-EBG as a preference for small sets of maximally general Horn clauses. In fact, the greedy algorithm of PROLOG-EBG is only a heuristic approximation to the exhaustive search algorithm that would be required to find the truly shortest set of maximally general Horn clauses. Nevertheless, the inductive bias of PROLOG-EBG can be approximately characterized in this fashion.

> **Approximate inductive bias of PROLOG-EBG:** The domain theory B, plus a preference for small sets of maximally general Horn clauses.

The most important point here is that the inductive bias of PROLOG-EBG—the policy by which it generalizes beyond the training data—is largely determined by the input domain theory. This lies in stark contrast to most of the other learning algorithms we have discussed (e.g., neural networks, decision tree learning), in which the inductive bias is a fixed property of the learning algorithm, typically determined by the syntax of its hypothesis representation. Why is it important that the inductive bias be an input parameter rather than a fixed property of the learner? Because, as we have discussed in Chapter 2 and elsewhere, there is no universally effective inductive bias and because bias-free learning is futile. Therefore, any attempt to develop a general-purpose learning method must at minimum allow the inductive bias to vary with the learning problem at hand. On a more practical level, in many tasks it is quite natural to input domain-specific knowledge (e.g., the knowledge about *Weight* in the *SafeToStack* example) to influence how the learner will generalize beyond the training data. In contrast, it is less natural to "implement" an appropriate bias by restricting the syntactic form of the hypotheses (e.g., prefer short decision trees). Finally, if we consider the larger issue of how an autonomous agent may improve its learning capabilities over time, then it is attractive to have a learning algorithm whose generalization capabilities improve as it acquires more knowledge of its domain.

11.3.4 Knowledge Level Learning

As pointed out in Equation (11.2), the hypothesis h output by PROLOG-EBG follows deductively from the domain theory B and training data D. In fact, by examining the PROLOG-EBG algorithm it is easy to see that h follows directly from B alone, independent of D. One way to see this is to imagine an algorithm that we might

call LEMMA-ENUMERATOR. The LEMMA-ENUMERATOR algorithm simply enumerates all proof trees that conclude the target concept based on assertions in the domain theory B. For each such proof tree, LEMMA-ENUMERATOR calculates the weakest preimage and constructs a Horn clause, in the same fashion as PROLOG-EBG. The only difference between LEMMA-ENUMERATOR and PROLOG-EBG is that LEMMA-ENUMERATOR ignores the training data and enumerates all proof trees.

Notice LEMMA-ENUMERATOR will output a superset of the Horn clauses output by PROLOG-EBG. Given this fact, several questions arise. First, if its hypotheses follow from the domain theory alone, then what is the role of training data in PROLOG-EBG? The answer is that training examples focus the PROLOG-EBG algorithm on generating rules that cover the distribution of instances that occur in practice. In our original chess example, for instance, the set of all possible lemmas is huge, whereas the set of chess positions that occur in normal play is only a small fraction of those that are syntactically possible. Therefore, by focusing only on training examples encountered in practice, the program is likely to develop a smaller, more relevant set of rules than if it attempted to enumerate all possible lemmas about chess.

The second question that arises is whether PROLOG-EBG can ever learn a hypothesis that goes beyond the knowledge that is already implicit in the domain theory. Put another way, will it ever learn to classify an instance that could not be classified by the original domain theory (assuming a theorem prover with unbounded computational resources)? Unfortunately, it will not. If $B \vdash h$, then any classification entailed by h will also be entailed by B. Is this an inherent limitation of analytical or deductive learning methods? No, it is not, as illustrated by the following example.

To produce an instance of deductive learning in which the learned hypothesis h entails conclusions that are not entailed by B, we must create an example where $B \not\vdash h$ but where $D \wedge B \vdash h$ (recall the constraint given by Equation (11.2)). One interesting case is when B contains assertions such as "If x satisfies the target concept, then so will $g(x)$." Taken alone, this assertion does not entail the classification of any instances. However, once we observe a positive example, it allows generalizing deductively to other unseen instances. For example, consider learning the *PlayTennis* target concept, describing the days on which our friend Ross would like to play tennis. Imagine that each day is described only by the single attribute *Humidity*, and the domain theory B includes the single assertion "If Ross likes to play tennis when the humidity is x, then he will also like to play tennis when the humidity is lower than x," which can be stated more formally as

$$(\forall x) \quad \text{IF } ((PlayTennis = Yes) \leftarrow (Humidity = x))$$
$$\text{THEN } ((PlayTennis = Yes) \leftarrow (Humidity \leq x))$$

Note that this domain theory does not entail any conclusions regarding which instances are positive or negative instances of *PlayTennis*. However, once the learner observes a positive example day for which $Humidity = .30$, the domain theory together with this positive example entails the following general hypothe-

sis h:

$$(PlayTennis = Yes) \leftarrow (Humidity \leq .30)$$

To summarize, this example illustrates a situation where $B \not\vdash h$, but where $B \wedge D \vdash h$. The learned hypothesis in this case entails predictions that are not entailed by the domain theory alone. The phrase *knowledge-level learning* is sometimes used to refer to this type of learning, in which the learned hypothesis entails predictions that go beyond those entailed by the domain theory. The set of all predictions entailed by a set of assertions Y is often called the *deductive closure* of Y. The key distinction here is that in knowledge-level learning the deductive closure of B is a proper subset of the deductive closure of $B + h$.

A second example of knowledge-level analytical learning is provided by considering a type of assertions known as *determinations*, which have been explored in detail by Russell (1989) and others. Determinations assert that some attribute of the instance is fully determined by certain other attributes, without specifying the exact nature of the dependence. For example, consider learning the target concept "people who speak Portuguese," and imagine we are given as a domain theory the single determination assertion "the language spoken by a person is determined by their nationality." Taken alone, this domain theory does not enable us to classify any instances as positive or negative. However, if we observe that "Joe, a 23-year-old left-handed Brazilian, speaks Portuguese," then we can conclude from this positive example and the domain theory that "all Brazilians speak Portuguese."

Both of these examples illustrate how deductive learning can produce output hypotheses that are not entailed by the domain theory alone. In both of these cases, the output hypothesis h satisfies $B \wedge D \vdash h$, but does not satisfy $B \vdash h$. In both cases, the learner *deduces* a justified hypothesis that does not follow from either the domain theory alone or the training data alone.

11.4 EXPLANATION-BASED LEARNING OF SEARCH CONTROL KNOWLEDGE

As noted above, the practical applicability of the PROLOG-EBG algorithm is restricted by its requirement that the domain theory be correct and complete. One important class of learning problems where this requirement is easily satisfied is learning to speed up complex search programs. In fact, the largest scale attempts to apply explanation-based learning have addressed the problem of learning to control search, or what is sometimes called "speedup" learning. For example, playing games such as chess involves searching through a vast space of possible moves and board positions to find the best move. Many practical scheduling and optimization problems are easily formulated as large search problems, in which the task is to find some move toward the goal state. In such problems the definitions of the legal search operators, together with the definition of the search objective, provide a complete and correct domain theory for learning search control knowledge.

Exactly how should we formulate the problem of learning search control so that we can apply explanation-based learning? Consider a general search problem where S is the set of possible search states, O is a set of legal search operators that transform one search state into another, and G is a predicate defined over S that indicates which states are goal states. The problem in general is to find a sequence of operators that will transform an arbitrary initial state s_i to some final state s_f that satisfies the goal predicate G. One way to formulate the learning problem is to have our system learn a separate target concept for each of the operators in O. In particular, for each operator o in O it might attempt to learn the target concept "the set of states for which o leads toward a goal state." Of course the exact choice of which target concepts to learn depends on the internal structure of problem solver that must use this learned knowledge. For example, if the problem solver is a means-ends planning system that works by establishing and solving subgoals, then we might instead wish to learn target concepts such as "the set of planning states in which subgoals of type A should be solved before subgoals of type B."

One system that employs explanation-based learning to improve its search is PRODIGY (Carbonell et al. 1990). PRODIGY is a domain-independent planning system that accepts the definition of a problem domain in terms of the state space S and operators O. It then solves problems of the form "find a sequence of operators that leads from initial state s_i to a state that satisfies goal predicate G." PRODIGY uses a means-ends planner that decomposes problems into subgoals, solves them, then combines their solutions into a solution for the full problem. Thus, during its search for problem solutions PRODIGY repeatedly faces questions such as "Which subgoal should be solved next?" and "Which operator should be considered for solving this subgoal?" Minton (1988) describes the integration of explanation-based learning into PRODIGY by defining a set of target concepts appropriate for these kinds of control decisions that it repeatedly confronts. For example, one target concept is "the set of states in which subgoal A should be solved before subgoal B." An example of a rule learned by PRODIGY for this target concept in a simple block-stacking problem domain is

IF	One subgoal to be solved is $On(x, y)$, and
	One subgoal to be solved is $On(y, z)$
THEN	Solve the subgoal $On(y, z)$ before $On(x, y)$

To understand this rule, consider again the simple block stacking problem illustrated in Figure 9.3. In the problem illustrated by that figure, the goal is to stack the blocks so that they spell the word "universal." PRODIGY would decompose this problem into several subgoals to be achieved, including $On(U, N)$, $On(N, I)$, etc. Notice the above rule matches the subgoals $On(U, N)$ and $On(N, I)$, and recommends solving the subproblem $On(N, I)$ before solving $On(U, N)$. The justification for this rule (and the explanation used by PRODIGY to learn the rule) is that if we solve the subgoals in the reverse sequence, we will encounter a conflict in which we must undo the solution to the $On(U, N)$ subgoal in order to achieve the other subgoal $On(N, I)$. PRODIGY learns by first encountering such a conflict, then

explaining to itself the reason for this conflict and creating a rule such as the one above. The net effect is that PRODIGY uses domain-independent knowledge about possible subgoal conflicts, together with domain-specific knowledge of specific operators (e.g., the fact that the robot can pick up only one block at a time), to learn useful domain-specific planning rules such as the one illustrated above.

The use of explanation-based learning to acquire control knowledge for PRODIGY has been demonstrated in a variety of problem domains including the simple block-stacking problem above, as well as more complex scheduling and planning problems. Minton (1988) reports experiments in three problem domains, in which the learned control rules improve problem-solving efficiency by a factor of two to four. Furthermore, the performance of these learned rules is comparable to that of handwritten rules across these three problem domains. Minton also describes a number of extensions to the basic explanation-based learning procedure that improve its effectiveness for learning control knowledge. These include methods for simplifying learned rules and for removing learned rules whose benefits are smaller than their cost.

A second example of a general problem-solving architecture that incorporates a form of explanation-based learning is the SOAR system (Laird et al. 1986; Newell 1990). SOAR supports a broad variety of problem-solving strategies that subsumes PRODIGY's means-ends planning strategy. Like PRODIGY, however, SOAR learns by explaining situations in which its current search strategy leads to inefficiencies. When it encounters a search choice for which it does not have a definite answer (e.g., which operator to apply next) SOAR reflects on this search impasse, using weak methods such as generate-and-test to determine the correct course of action. The reasoning used to resolve this impasse can be interpreted as an explanation for how to resolve similar impasses in the future. SOAR uses a variant of explanation-based learning called *chunking* to extract the general conditions under which the same explanation applies. SOAR has been applied in a great number of problem domains and has also been proposed as a psychologically plausible model of human learning processes (see Newell 1990).

PRODIGY and SOAR demonstrate that explanation-based learning methods can be successfully applied to acquire search control knowledge in a variety of problem domains. Nevertheless, many or most heuristic search programs still use numerical evaluation functions similar to the one described in Chapter 1, rather than rules acquired by explanation-based learning. What is the reason for this? In fact, there are significant practical problems with applying EBL to learning search control. First, in many cases the number of control rules that must be learned is very large (e.g., many thousands of rules). As the system learns more and more control rules to improve its search, it must pay a larger and larger cost at each step to match this set of rules against the current search state. Note this problem is not specific to explanation-based learning; it will occur for any system that represents its learned knowledge by a growing set of rules. Efficient algorithms for matching rules can alleviate this problem, but not eliminate it completely. Minton (1988) discusses strategies for empirically estimating the computational cost and benefit of each rule, learning rules only when the estimated benefits outweigh the estimated costs

and deleting rules later found to have negative utility. He describes how using this kind of *utility analysis* to determine what should be learned and what should be forgotten significantly enhances the effectiveness of explanation-based learning in PRODIGY. For example, in a series of robot block-stacking problems, PRODIGY encountered 328 opportunities for learning a new rule, but chose to exploit only 69 of these, and eventually reduced the learned rules to a set of 19, once low-utility rules were eliminated. Tambe et al. (1990) and Doorenbos (1993) discuss how to identify types of rules that will be particularly costly to match, as well as methods for re-expressing such rules in more efficient forms and methods for optimizing rule-matching algorithms. Doorenbos (1993) describes how these methods enabled SOAR to efficiently match a set of 100,000 learned rules in one problem domain, without a significant increase in the cost of matching rules per state.

A second practical problem with applying explanation-based learning to learning search control is that in many cases it is intractable even to construct the explanations for the desired target concept. For example, in chess we might wish to learn a target concept such as "states for which operator A leads toward the optimal solution." Unfortunately, to prove or explain why A leads toward the optimal solution requires explaining that every alternative operator leads to a less optimal outcome. This typically requires effort exponential in the search depth. Chien (1993) and Tadepalli (1990) explore methods for "lazy" or "incremental" explanation, in which heuristics are used to produce partial and approximate, but tractable, explanations. Rules are extracted from these imperfect explanations as though the explanations were perfect. Of course these learned rules may be incorrect due to the incomplete explanations. The system accommodates this by monitoring the performance of the rule on subsequent cases. If the rule subsequently makes an error, then the original explanation is incrementally elaborated to cover the new case, and a more refined rule is extracted from this incrementally improved explanation.

Many additional research efforts have explored the use of explanation-based learning for improving the efficiency of search-based problem solvers (for example, Mitchell 1981; Silver 1983; Shavlik 1990; Mahadevan et al. 1993; Gervasio and DeJong 1994; DeJong 1994). Bennett and DeJong (1996) explore explanation-based learning for robot planning problems where the system has an imperfect domain theory that describes its world and actions. Dietterich and Flann (1995) explore the integration of explanation-based learning with reinforcement learning methods discussed in Chapter 13. Mitchell and Thrun (1993) describe the application of an explanation-based neural network learning method (see the EBNN algorithm discussed in Chapter 12) to reinforcement learning problems.

11.5 SUMMARY AND FURTHER READING

The main points of this chapter include:

- In contrast to purely inductive learning methods that seek a hypothesis to fit the training data, purely analytical learning methods seek a hypothesis

that fits the learner's prior knowledge and covers the training examples. Humans often make use of prior knowledge to guide the formation of new hypotheses. This chapter examines purely analytical learning methods. The next chapter examines combined inductive-analytical learning.

- Explanation-based learning is a form of analytical learning in which the learner processes each novel training example by (1) explaining the observed target value for this example in terms of the domain theory, (2) analyzing this explanation to determine the general conditions under which the explanation holds, and (3) refining its hypothesis to incorporate these general conditions.
- PROLOG-EBG is an explanation-based learning algorithm that uses first-order Horn clauses to represent both its domain theory and its learned hypotheses. In PROLOG-EBG an explanation is a PROLOG proof, and the hypothesis extracted from the explanation is the weakest preimage of this proof. As a result, the hypotheses output by PROLOG-EBG follow deductively from its domain theory.
- Analytical learning methods such as PROLOG-EBG construct useful intermediate features as a side effect of analyzing individual training examples. This analytical approach to feature generation complements the statistically based generation of intermediate features (eg., hidden unit features) in inductive methods such as BACKPROPAGATION.
- Although PROLOG-EBG does not produce hypotheses that extend the deductive closure of its domain theory, other deductive learning procedures can. For example, a domain theory containing determination assertions (e.g., "nationality determines language") can be used together with observed data to deductively infer hypotheses that go beyond the deductive closure of the domain theory.
- One important class of problems for which a correct and complete domain theory can be found is the class of large state-space search problems. Systems such as PRODIGY and SOAR have demonstrated the utility of explanation-based learning methods for automatically acquiring effective search control knowledge that speeds up problem solving in subsequent cases.
- Despite the apparent usefulness of explanation-based learning methods in humans, purely deductive implementations such as PROLOG-EBG suffer the disadvantage that the output hypothesis is only as correct as the domain theory. In the next chapter we examine approaches that combine inductive and analytical learning methods in order to learn effectively from imperfect domain theories and limited training data.

The roots of analytical learning methods can be traced to early work by Fikes et al. (1972) on learning macro-operators through analysis of operators in ABSTRIPS and to somewhat later work by Soloway (1977) on the use of explicit prior knowledge in learning. Explanation-based learning methods similar to those discussed in this chapter first appeared in a number of systems developed during the early 1980s, including DeJong (1981); Mitchell (1981); Winston et al.

(1983); and Silver (1983). DeJong and Mooney (1986) and Mitchell et al. (1986) provided general descriptions of the explanation-based learning paradigm, which helped spur a burst of research on this topic during the late 1980s. A collection of research on explanation-based learning performed at the University of Illinois is described by DeJong (1993), including algorithms that modify the structure of the explanation in order to correctly generalize iterative and temporal explanations. More recent research has focused on extending explanation-based methods to accommodate imperfect domain theories and to incorporate inductive together with analytical learning (see Chapter 12). An edited collection exploring the role of goals and prior knowledge in human and machine learning is provided by Ram and Leake (1995), and a recent overview of explanation-based learning is given by DeJong (1997).

The most serious attempts to employ explanation-based learning with perfect domain theories have been in the area of learning search control, or "speedup" learning. The SOAR system described by Laird et al. (1986) and the PRODIGY system described by Carbonell et al. (1990) are among the most developed systems that use explanation-based learning methods for learning in problem solving. Rosenbloom and Laird (1986) discuss the close relationship between SOAR's learning method (called "chunking") and other explanation-based learning methods. More recently, Dietterich and Flann (1995) have explored the combination of explanation-based learning with reinforcement learning methods for learning search control.

While our primary purpose here is to study machine learning algorithms, it is interesting to note that experimental studies of human learning provide support for the conjecture that human learning is based on explanations. For example, Ahn et al. (1987) and Qin et al. (1992) summarize evidence supporting the conjecture that humans employ explanation-based learning processes. Wisniewski and Medin (1995) describe experimental studies of human learning that suggest a rich interplay between prior knowledge and observed data to influence the learning process. Kotovsky and Baillargeon (1994) describe experiments that suggest even 11-month-old infants build on prior knowledge as they learn.

The analysis performed in explanation-based learning is similar to certain kinds of program optimization methods used for PROLOG programs, such as partial evaluation; van Harmelen and Bundy (1988) provide one discussion of the relationship.

EXERCISES

11.1. Consider the problem of learning the target concept "pairs of people who live in the same house," denoted by the predicate $HouseMates(x, y)$. Below is a positive example of the concept.

$HouseMates(Joe, Sue)$

$Person(Joe)$	$Person(Sue)$
$Sex(Joe, Male)$	$Sex(Sue, Female)$
$HairColor(Joe, Black)$	$HairColor(Sue, Brown)$

$Height(Joe, Short)$	$Height(Sue, Short)$
$Nationality(Joe, US)$	$Nationality(Sue, US)$
$Mother(Joe, Mary)$	$Mother(Sue, Mary)$
$Age(Joe, 8)$	$Age(Sue, 6)$

The following domain theory is helpful for acquiring the *HouseMates* concept:

$$HouseMates(x, y) \leftarrow InSameFamily(x, y)$$
$$HouseMates(x, y) \leftarrow FraternityBrothers(x, y)$$
$$InSameFamily(x, y) \leftarrow Married(x, y)$$
$$InSameFamily(x, y) \leftarrow Youngster(x) \wedge Youngster(y) \wedge SameMother(x, y)$$
$$SameMother(x, y) \leftarrow Mother(x, z) \wedge Mother(y, z)$$
$$Youngster(x) \leftarrow Age(x, a) \wedge LessThan(a, 10)$$

Apply the PROLOG-EBG algorithm to the task of generalizing from the above instance, using the above domain theory. In particular,

(*a*) Show a hand-trace of the PROLOG-EBG algorithm applied to this problem; that is, show the explanation generated for the training instance, show the result of regressing the target concept through this explanation, and show the resulting Horn clause rule.

(*b*) Suppose that the target concept is "people who live with Joe" instead of "pairs of people who live together." Write down this target concept in terms of the above formalism. Assuming the same training instance and domain theory as before, what Horn clause rule will PROLOG-EBG produce for this new target concept?

11.2. As noted in Section 11.3.1, PROLOG-EBG can construct useful new features that are not explicit features of the instances, but that are defined in terms of the explicit features and that are useful for describing the appropriate generalization. These features are derived as a side effect of analyzing the training example explanation. A second method for deriving useful features is the BACKPROPAGATION algorithm for multilayer neural networks, in which new features are learned by the hidden units based on the statistical properties of a large number of examples. Can you suggest a way in which one might combine these analytical and inductive approaches to generating new features? (Warning: This is an open research problem.)

REFERENCES

Ahn, W., Mooney, R. J., Brewer, W. F., & DeJong, G. F. (1987). Schema acquisition from one example: Psychological evidence for explanation-based learning. *Ninth Annual Conference of the Cognitive Science Society* (pp. 50–57). Hillsdale, NJ: Lawrence Erlbaum Associates.

Bennett, S. W., & DeJong, G. F. (1996). Real-world robotics: Learning to plan for robust execution. *Machine Learning*, 23, 121.

Carbonell, J., Knoblock, C., & Minton, S. (1990). PRODIGY: An integrated architecture for planning and learning. In K. VanLehn (Ed.), *Architectures for Intelligence*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Chien, S. (1993). NONMON: Learning with recoverable simplifications. In G. DeJong (Ed.), *Investigating explanation-based learning* (pp. 410–434). Boston, MA: Kluwer Academic Publishers.

Davies, T. R., and Russell, S. J. (1987). A logical approach to reasoning by analogy. *Proceedings of the 10th International Joint Conference on Artificial Intelligence* (pp. 264–270). San Mateo, CA: Morgan Kaufmann.

DeJong, G. (1981). Generalizations based on explanations. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 67–70).

DeJong, G., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, 1(2), 145–176.

DeJong, G. (Ed.). (1993). *Investigating explanation-based learning*. Boston, MA: Kluwer Academic Publishers.

DeJong, G. (1994). Learning to plan in continuous domains. *Artificial Intelligence*, 64(1), 71–141.

DeJong, G. (1997). Explanation-based learning. In A. Tucker (Ed.), *The Computer Science and Engineering Handbook* (pp. 499–520). Boca Raton, FL: CRC Press.

Dietterich, T. G., Flann, N. S. (1995). Explanation-based learning and reinforcement learning: A unified view. *Proceedings of the 12th International Conference on Machine Learning* (pp. 176–184). San Mateo, CA: Morgan Kaufmann.

Doorenbos, R. E. (1993). Matching 100,000 learned rules. *Proceedings of the Eleventh National Conference on Artificial Intelligence* (pp. 290–296). AAAI Press/MIT Press.

Fikes, R., Hart, P., & Nilsson, N. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4), 251–288.

Fisher, D., Subramanian, D., & Tadepalli, P. (1992). An overview of current research on knowledge compilation and speedup learning. *Proceedings of the Second International Workshop on Knowledge Compilation and Speedup Learning*.

Flann, N. S., & Dietterich, T. G. (1989). A study of explanation-based methods for inductive learning. *Machine Learning*, 4, 187–226.

Gervasio, M. T., & DeJong, G. F. (1994). An incremental learning approach to completable planning. *Proceedings of the Eleventh International Conference on Machine Learning*, New Brunswick, NJ. San Mateo, CA: Morgan Kaufmann.

van Harmelen, F., & Bundy, A. (1988). Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36(3), 401–412.

Kedar-Cabelli, S., & McCarty, T. (1987). Explanation-based generalization as resolution theorem proving. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 383–389). San Francisco: Morgan Kaufmann.

Kotovsky, L., & Baillargeon, R. (1994). Calibration-based reasoning about collision events in 11-month-old infants. *Cognition*, 51, 107–129.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1, 11.

Mahadevan, S., Mitchell, T., Mostow, D. J., Steinberg, L., & Tadepalli, P. (1993). An apprentice-based approach to knowledge acquisition. In S. Mahadevan, T. Mitchell, D. J. Mostow, L. Steinberg, & P. Tadepalli (Eds.), *Artificial Intelligence*, 64(1), 1–52.

Minton, S. (1988). *Learning search control knowledge: An explanation-based approach*. Boston, MA: Kluwer Academic Publishers.

Minton, S., Carbonell, J., Knoblock, C., Kuokka, D., Etzioni, O., & Gil, Y. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40, 63–118.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42, 363–391.

Mitchell, T. M. (1981). *Toward combining empirical and analytical methods for inferring heuristics* (Technical Report LCSR-TR-27), Rutgers Computer Science Department. (Also reprinted in A. Elithorn & R. Banerji (Eds), *Artificial and Human Intelligence*. North-Holland, 1984.)

Mitchell, T. M. (1983). Learning and problem-solving. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. San Francisco: Morgan Kaufmann.

Mitchell, T. M., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 47–80.

Mitchell, T. M. (1990). Becoming increasingly reactive. *Proceedings of the Eighth National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press.

Mitchell, T. M., & Thrun, S. B. (1993). Explanation-based neural network learning for robot control. In S. Hanson et al. (Eds.), *Advances in neural information processing systems 5* (pp. 287–294). San Mateo, CA: Morgan-Kaufmann Press.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Qin, Y., Mitchell, T., & Simon, H. (1992). Using explanation-based generalization to simulate human learning from examples and learning by doing. *Proceedings of the Florida AI Research Symposium* (pp. 235–239).

Ram, A., & Leake, D. B. (Eds.). (1995). *Goal-driven learning*. Cambridge, MA: MIT Press.

Rosenbloom, P., & Laird, J. (1986). Mapping explanation-based generalization onto SOAR. *Fifth National Conference on Artificial Intelligence* (pp. 561–567). AAAI Press.

Russell, S. (1989). *The use of knowledge in analogy and induction*. San Francisco: Morgan Kaufmann.

Shavlik, J. W. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5, 39.

Silver, B. (1983). Learning equation solving methods from worked examples. *Proceedings of the 1983 International Workshop on Machine Learning* (pp. 99–104). CS Department, University of Illinois at Urbana-Champaign.

Silver, B. (1986). Precondition analysis: Learning control information. In R. Michalski et al. (Eds.), *Machine Learning: An AI approach* (pp. 647–670). San Mateo, CA: Morgan Kaufmann.

Soloway, E. (1977). *Knowledge directed learning using multiple levels of description* (Ph.D. thesis). University of Massachusetts, Amherst.

Tadepalli, P. (1990). *Tractable learning and planning in games* (Technical report ML-TR-31) (Ph.D. dissertation). Rutgers University Computer Science Department.

Tambe, M., Newell, A., & Rosenbloom, P. S. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5(4), 299–348.

Waldinger, R. (1977). Achieving several goals simultaneously. In E. Elcock & D. Michie (Eds.), *Machine Intelligence 8*. London: Ellis Horwood Ltd.

Winston, P., Binford, T., Katz, B., & Lowry, M. (1983). Learning physical descriptions from functional definitions, examples, and precedents. *Proceedings of the National Conference on Artificial Intelligence* (pp. 433–439). San Mateo, CA: Morgan Kaufmann.

Wisniewski, E. J., & Medin, D. L. (1995). Harpoons and long sticks: The interaction of theory and similarity in rule induction. In A. Ram & D. B. Leake (Eds.), *Goal-driven learning* (pp. 177–210). Cambridge, MA: MIT Press.