19 KNOWLEDGE IN LEARNING

In which we examine the problem of learning when you know something already.

In all of the approaches to learning described in the previous chapter, the idea is to construct a function that has the input–output behavior observed in the data. In each case, the learning methods can be understood as searching a hypothesis space to find a suitable function, starting from only a very basic assumption about the form of the function, such as "second-degree polynomial" or "decision tree" and perhaps a preference for simpler hypotheses. Doing this amounts to saying that before you can learn something new, you must first forget (almost) everything you know. In this chapter, we study learning methods that can take advantage of **prior knowledge** about the world. In most cases, the prior knowledge is represented as general first-order logical theories; thus for the first time we bring together the work on knowledge representation and learning.

PRIOR KNOWLEDGE

19.1 A LOGICAL FORMULATION OF LEARNING

Chapter 18 defined pure inductive learning as a process of finding a hypothesis that agrees with the observed examples. Here, we specialize this definition to the case where the hypothesis is represented by a set of logical sentences. Example descriptions and classifications will also be logical sentences, and a new example can be classified by inferring a classification sentence from the hypothesis and the example description. This approach allows for incremental construction of hypotheses, one sentence at a time. It also allows for prior knowledge, because sentences that are already known can assist in the classification of new examples. The logical formulation of learning may seem like a lot of extra work at first, but it turns out to clarify many of the issues in learning. It enables us to go well beyond the simple learning methods of Chapter 18 by using the full power of logical inference in the service of learning.

19.1.1 Examples and hypotheses

Recall from Chapter 18 the restaurant learning problem: learning a rule for deciding whether to wait for a table. Examples were described by **attributes** such as $Alternate$, Bar, Fri/Sat, and so on. In a logical setting, an example is described by a logical sentence; the attributes become unary predicates. Let us generically call the ith example X_i. For instance, the first example from Figure 18.3 (page 700) is described by the sentences

$$Alternate(X_1) \wedge \neg Bar(X_1) \wedge \neg Fri/Sat(X_1) \wedge Hungry(X_1) \wedge \ldots$$

We will use the notation $D_i(X_i)$ to refer to the description of X_i, where D_i can be any logical expression taking a single argument. The classification of the example is given by a literal using the goal predicate, in this case

$$WillWait(X_1) \qquad \text{or} \qquad \neg\, WillWait(X_1)\ .$$

The complete training set can thus be expressed as the conjunction of all the example descriptions and goal literals.

The aim of inductive learning in general is to find a hypothesis that classifies the examples well and generalizes well to new examples. Here we are concerned with hypotheses expressed in logic; each hypothesis h_j will have the form

$$\forall\, x \;\; Goal(x) \;\Leftrightarrow\; C_j(x)\ ,$$

where $C_j(x)$ is a candidate definition—some expression involving the attribute predicates. For example, a decision tree can be interpreted as a logical expression of this form. Thus, the tree in Figure 18.6 (page 702) expresses the following logical definition (which we will call h_r for future reference):

$$\begin{aligned} \forall\, r \;\; WillWait(r) \;\Leftrightarrow\; & Patrons(r, Some) \\ \vee\; & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, French) \\ \vee\; & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, Thai) \\ & \qquad \wedge Fri/Sat(r) \\ \vee\; & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, Burger)\ . \end{aligned} \tag{19.1}$$

EXTENSION

Each hypothesis predicts that a certain set of examples—namely, those that satisfy its candidate definition—will be examples of the goal predicate. This set is called the **extension** of the predicate. Two hypotheses with different extensions are therefore logically inconsistent with each other, because they disagree on their predictions for at least one example. If they have the same extension, they are logically equivalent.

The hypothesis space $\mathcal{H}$ is the set of all hypotheses $\{h_1, \ldots, h_n\}$ that the learning algorithm is designed to entertain. For example, the DECISION-TREE-LEARNING algorithm can entertain any decision tree hypothesis defined in terms of the attributes provided; its hypothesis space therefore consists of all these decision trees. Presumably, the learning algorithm believes that one of the hypotheses is correct; that is, it believes the sentence

$$h_1 \vee h_2 \vee h_3 \vee \ldots \vee h_n\ . \tag{19.2}$$

As the examples arrive, hypotheses that are not **consistent** with the examples can be ruled out. Let us examine this notion of consistency more carefully. Obviously, if hypothesis h_j is consistent with the entire training set, it has to be consistent with each example in the training set. What would it mean for it to be inconsistent with an example? There are two possible ways that this can happen:

FALSE NEGATIVE

- An example can be a **false negative** for the hypothesis, if the hypothesis says it should be negative but in fact it is positive. For instance, the new example X_{13} described by $Patrons(X_{13}, Full) \wedge \neg Hungry(X_{13}) \wedge \ldots \wedge WillWait(X_{13})$ would be a false negative for the hypothesis h_r given earlier. From h_r and the example description, we can deduce both $WillWait(X_{13})$, which is what the example says, and $\neg WillWait(X_{13})$, which is what the hypothesis predicts. The hypothesis and the example are therefore logically inconsistent.

FALSE POSITIVE

- An example can be a **false positive** for the hypothesis, if the hypothesis says it should be positive but in fact it is negative.[1]

If an example is a false positive or false negative for a hypothesis, then the example and the hypothesis are logically inconsistent with each other. Assuming that the example is a correct observation of fact, then the hypothesis can be ruled out. Logically, this is exactly analogous to the resolution rule of inference (see Chapter 9), where the disjunction of hypotheses corresponds to a clause and the example corresponds to a literal that resolves against one of the literals in the clause. An ordinary logical inference system therefore could, in principle, learn from the example by eliminating one or more hypotheses. Suppose, for example, that the example is denoted by the sentence I_1, and the hypothesis space is $h_1 \vee h_2 \vee h_3 \vee h_4$. Then if I_1 is inconsistent with h_2 and h_3, the logical inference system can deduce the new hypothesis space $h_1 \vee h_4$.

We therefore can characterize inductive learning in a logical setting as a process of gradually eliminating hypotheses that are inconsistent with the examples, narrowing down the possibilities. Because the hypothesis space is usually vast (or even infinite in the case of first-order logic), we do not recommend trying to build a learning system using resolution-based theorem proving and a complete enumeration of the hypothesis space. Instead, we will describe two approaches that find logically consistent hypotheses with much less effort.

19.1.2 Current-best-hypothesis search

CURRENT-BEST-HYPOTHESIS

The idea behind **current-best-hypothesis** search is to maintain a single hypothesis, and to adjust it as new examples arrive in order to maintain consistency. The basic algorithm was described by John Stuart Mill (1843), and may well have appeared even earlier.

Suppose we have some hypothesis such as h_r, of which we have grown quite fond. As long as each new example is consistent, we need do nothing. Then along comes a false negative example, X_{13}. What do we do? Figure 19.1(a) shows h_r schematically as a region: everything inside the rectangle is part of the extension of h_r. The examples that have actually been seen so far are shown as "+" or "–", and we see that h_r correctly categorizes all the examples as positive or negative examples of $WillWait$. In Figure 19.1(b), a new example (circled) is a false negative: the hypothesis says it should be negative but it is actually positive. The extension of the hypothesis must be increased to include it. This is called **generalization**; one possible generalization is shown in Figure 19.1(c). Then in Figure 19.1(d), we see a false positive: the hypothesis says the new example (circled) should be positive, but it actually is

GENERALIZATION

[1] The terms "false positive" and "false negative" are used in medicine to describe erroneous results from lab tests. A result is a false positive if it indicates that the patient has the disease when in fact no disease is present.

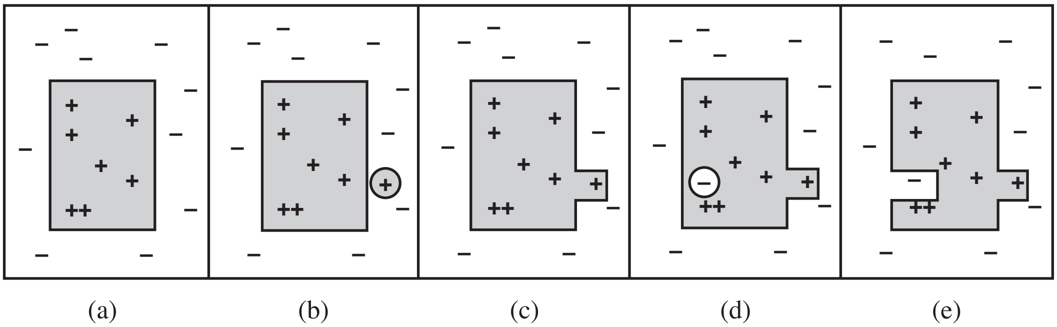


Figure 19.1 (a) A consistent hypothesis. (b) A false negative. (c) The hypothesis is generalized. (d) A false positive. (e) The hypothesis is specialized.

```
function CURRENT-BEST-LEARNING($examples$, $h$) returns a hypothesis or fail

  if $examples$ is empty then
     return $h$
  $e \leftarrow$ FIRST($examples$)
  if $e$ is consistent with $h$ then
     return CURRENT-BEST-LEARNING(REST($examples$), $h$)
  else if $e$ is a false positive for $h$ then
     for each $h'$ in  specializations of $h$ consistent with $examples$ seen so far do
        $h'' \leftarrow$ CURRENT-BEST-LEARNING(REST($examples$), $h'$)
        if $h'' \neq fail$ then return $h''$
  else if $e$ is a false negative for $h$ then
     for each $h'$ in generalizations of $h$ consistent with $examples$ seen so far do
        $h'' \leftarrow$ CURRENT-BEST-LEARNING(REST($examples$), $h'$)
        if $h'' \neq fail$ then return $h''$
  return $fail$
```

Figure 19.2 The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis that fits all the examples and backtracks when no consistent specialization/generalization can be found. To start the algorithm, any hypothesis can be passed in; it will be specialized or gneralized as needed.

SPECIALIZATION

negative. The extension of the hypothesis must be decreased to exclude the example. This is called **specialization**; in Figure 19.1(e) we see one possible specialization of the hypothesis. The "more general than" and "more specific than" relations between hypotheses provide the logical structure on the hypothesis space that makes efficient search possible.

We can now specify the CURRENT-BEST-LEARNING algorithm, shown in Figure 19.2. Notice that each time we consider generalizing or specializing the hypothesis, we must check for consistency with the other examples, because an arbitrary increase/decrease in the extension might include/exclude previously seen negative/positive examples.

We have defined generalization and specialization as operations that change the *extension* of a hypothesis. Now we need to determine exactly how they can be implemented as syntactic operations that change the candidate definition associated with the hypothesis, so that a program can carry them out. This is done by first noting that generalization and specialization are also *logical* relationships between hypotheses. If hypothesis h_1, with definition C_1, is a generalization of hypothesis h_2 with definition C_2, then we must have

$$\forall x \;\; C_2(x) \;\Rightarrow\; C_1(x) \; .$$

Therefore in order to construct a generalization of h_2, we simply need to find a definition C_1 that is logically implied by C_2. This is easily done. For example, if $C_2(x)$ is $\mathit{Alternate}(x) \wedge \mathit{Patrons}(x, \mathit{Some})$, then one possible generalization is given by $C_1(x) \equiv \mathit{Patrons}(x, \mathit{Some})$. This is called **dropping conditions**. Intuitively, it generates a weaker definition and therefore allows a larger set of positive examples. There are a number of other generalization operations, depending on the language being operated on. Similarly, we can specialize a hypothesis by adding extra conditions to its candidate definition or by removing disjuncts from a disjunctive definition. Let us see how this works on the restaurant example, using the data in Figure 18.3.

DROPPING CONDITIONS

- The first example, X_1, is positive. The attribute $\mathit{Alternate}(X_1)$ is true, so let the initial hypothesis be

  $$h_1: \;\; \forall x \;\; \mathit{WillWait}(x) \;\Leftrightarrow\; \mathit{Alternate}(x) \; .$$

- The second example, X_2, is negative. h_1 predicts it to be positive, so it is a false positive. Therefore, we need to specialize h_1. This can be done by adding an extra condition that will rule out X_2, while continuing to classify X_1 as positive. One possibility is

  $$h_2: \;\; \forall x \;\; \mathit{WillWait}(x) \;\Leftrightarrow\; \mathit{Alternate}(x) \wedge \mathit{Patrons}(x, \mathit{Some}) \; .$$

- The third example, X_3, is positive. h_2 predicts it to be negative, so it is a false negative. Therefore, we need to generalize h_2. We drop the $\mathit{Alternate}$ condition, yielding

  $$h_3: \;\; \forall x \;\; \mathit{WillWait}(x) \;\Leftrightarrow\; \mathit{Patrons}(x, \mathit{Some}) \; .$$

- The fourth example, X_4, is positive. h_3 predicts it to be negative, so it is a false negative. We therefore need to generalize h_3. We cannot drop the $\mathit{Patrons}$ condition, because that would yield an all-inclusive hypothesis that would be inconsistent with X_2. One possibility is to add a disjunct:

  $$\begin{aligned} h_4: \;\; \forall x \;\; \mathit{WillWait}(x) \;\Leftrightarrow\; & \mathit{Patrons}(x, \mathit{Some}) \\ & \vee (\mathit{Patrons}(x, \mathit{Full}) \wedge \mathit{Fri/Sat}(x)) \; . \end{aligned}$$

Already, the hypothesis is starting to look reasonable. Obviously, there are other possibilities consistent with the first four examples; here are two of them:

$$h_4': \;\; \forall x \;\; \mathit{WillWait}(x) \;\Leftrightarrow\; \neg\, \mathit{WaitEstimate}(x, \mathit{30\text{-}60}) \; .$$

$$\begin{aligned} h_4'': \;\; \forall x \;\; \mathit{WillWait}(x) \;\Leftrightarrow\; & \mathit{Patrons}(x, \mathit{Some}) \\ & \vee (\mathit{Patrons}(x, \mathit{Full}) \wedge \mathit{WaitEstimate}(x, \mathit{10\text{-}30})) \; . \end{aligned}$$

The CURRENT-BEST-LEARNING algorithm is described nondeterministically, because at any point, there may be several possible specializations or generalizations that can be applied. The

```
function VERSION-SPACE-LEARNING($examples$) returns a version space
  local variables: $V$, the version space: the set of all hypotheses

  $V \leftarrow$ the set of all hypotheses
  for each example $e$ in $examples$ do
     if $V$ is not empty then $V \leftarrow$ VERSION-SPACE-UPDATE($V$, $e$)
  return $V$
```

```
function VERSION-SPACE-UPDATE($V$, $e$) returns an updated version space
  $V \leftarrow \{h \in V : h$ is consistent with $e\}$
```

Figure 19.3 The version space learning algorithm. It finds a subset of V that is consistent with all the $examples$.

choices that are made will not necessarily lead to the simplest hypothesis, and may lead to an unrecoverable situation where no simple modification of the hypothesis is consistent with all of the data. In such cases, the program must backtrack to a previous choice point.

The CURRENT-BEST-LEARNING algorithm and its variants have been used in many machine learning systems, starting with Patrick Winston's (1970) "arch-learning" program. With a large number of examples and a large space, however, some difficulties arise:

1. Checking all the previous examples over again for each modification is very expensive.
2. The search process may involve a great deal of backtracking. As we saw in Chapter 18, hypothesis space can be a doubly exponentially large place.

19.1.3 Least-commitment search

Backtracking arises because the current-best-hypothesis approach has to *choose* a particular hypothesis as its best guess even though it does not have enough data yet to be sure of the choice. What we can do instead is to keep around all and only those hypotheses that are consistent with all the data so far. Each new example will either have no effect or will get rid of some of the hypotheses. Recall that the original hypothesis space can be viewed as a disjunctive sentence

$$h_1 \vee h_2 \vee h_3 \ldots \vee h_n \,.$$

VERSION SPACE

CANDIDATE ELIMINATION

As various hypotheses are found to be inconsistent with the examples, this disjunction shrinks, retaining only those hypotheses not ruled out. Assuming that the original hypothesis space does in fact contain the right answer, the reduced disjunction must still contain the right answer because only incorrect hypotheses have been removed. The set of hypotheses remaining is called the **version space**, and the learning algorithm (sketched in Figure 19.3) is called the version space learning algorithm (also the **candidate elimination** algorithm).

One important property of this approach is that it is *incremental*: one never has to go back and reexamine the old examples. All remaining hypotheses are guaranteed to be consistent with them already. But there is an obvious problem. We already said that the

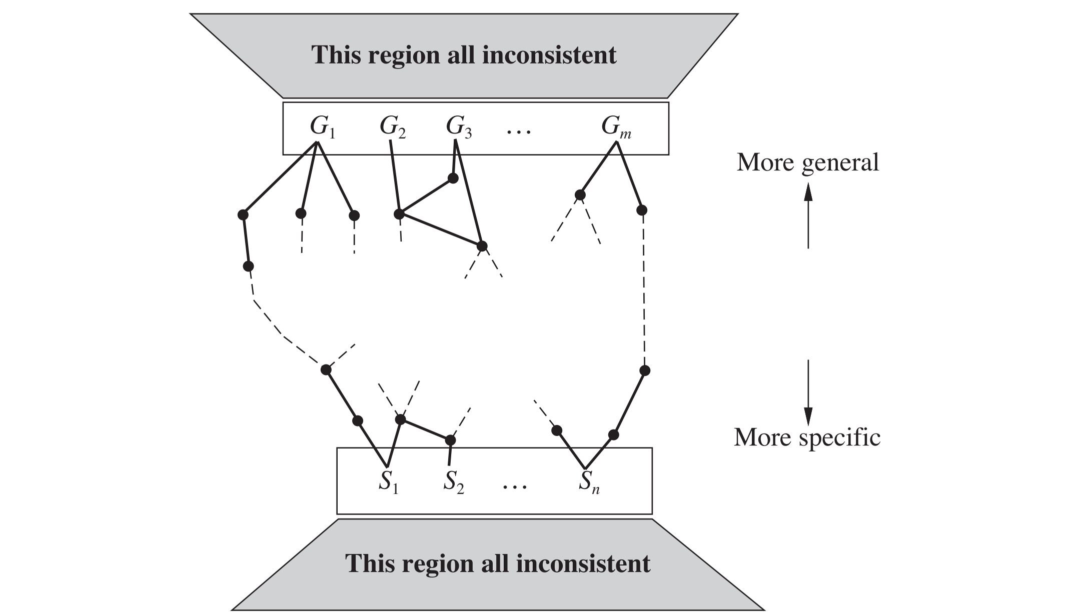

This region all inconsistent
G_1 G_2 G_3 … G_m
More general
More specific
S_1 S_2 … S_n
This region all inconsistent

Figure 19.4 The version space contains all hypotheses consistent with the examples.

hypothesis space is enormous, so how can we possibly write down this enormous disjunction?

The following simple analogy is very helpful. How do you represent all the real numbers between 1 and 2? After all, there are an infinite number of them! The answer is to use an interval representation that just specifies the boundaries of the set: [1,2]. It works because we have an *ordering* on the real numbers.

We also have an ordering on the hypothesis space, namely, generalization/specialization. This is a partial ordering, which means that each boundary will not be a point but rather a set of hypotheses called a **boundary set**. The great thing is that we can represent the entire version space using just two boundary sets: a most general boundary (the **G-set**) and a most specific boundary (the **S-set**). *Everything in between is guaranteed to be consistent with the examples.* Before we prove this, let us recap:

BOUNDARY SET

G-SET

S-SET

- The current version space is the set of hypotheses consistent with all the examples so far. It is represented by the S-set and G-set, each of which is a set of hypotheses.
- Every member of the S-set is consistent with all observations so far, and there are no consistent hypotheses that are more specific.
- Every member of the G-set is consistent with all observations so far, and there are no consistent hypotheses that are more general.

We want the initial version space (before any examples have been seen) to represent all possible hypotheses. We do this by setting the G-set to contain $True$ (the hypothesis that contains everything), and the S-set to contain $False$ (the hypothesis whose extension is empty).

Figure 19.4 shows the general structure of the boundary-set representation of the version space. To show that the representation is sufficient, we need the following two properties:

1. Every consistent hypothesis (other than those in the boundary sets) is more specific than some member of the G-set, and more general than some member of the S-set. (That is, there are no "stragglers" left outside.) This follows directly from the definitions of S and G. If there were a straggler h, then it would have to be no more specific than any member of G, in which case it belongs in G; or no more general than any member of S, in which case it belongs in S.
2. Every hypothesis more specific than some member of the G-set and more general than some member of the S-set is a consistent hypothesis. (That is, there are no "holes" between the boundaries.) Any h between S and G must reject all the negative examples rejected by each member of G (because it is more specific), and must accept all the positive examples accepted by any member of S (because it is more general). Thus, h must agree with all the examples, and therefore cannot be inconsistent. Figure 19.5 shows the situation: there are no known examples outside S but inside G, so any hypothesis in the gap must be consistent.

We have therefore shown that *if* S and G are maintained according to their definitions, then they provide a satisfactory representation of the version space. The only remaining problem is how to *update* S and G for a new example (the job of the VERSION-SPACE-UPDATE function). This may appear rather complicated at first, but from the definitions and with the help of Figure 19.4, it is not too hard to reconstruct the algorithm.

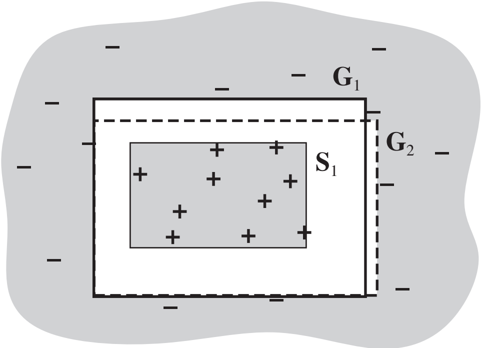


Figure 19.5 The extensions of the members of G and S. No known examples lie in between the two sets of boundaries.

We need to worry about the members S_i and G_i of the S- and G-sets. For each one, the new example may be a false positive or a false negative.

1. False positive for S_i: This means S_i is too general, but there are no consistent specializations of S_i (by definition), so we throw it out of the S-set.
2. False negative for S_i: This means S_i is too specific, so we replace it by all its immediate generalizations, provided they are more specific than some member of G.
3. False positive for G_i: This means G_i is too general, so we replace it by all its immediate specializations, provided they are more general than some member of S.

4. False negative for G_i: This means G_i is too specific, but there are no consistent generalizations of G_i (by definition) so we throw it out of the G-set.

We continue these operations for each new example until one of three things happens:

1. We have exactly one hypothesis left in the version space, in which case we return it as the unique hypothesis.
2. The version space *collapses*—either S or G becomes empty, indicating that there are no consistent hypotheses for the training set. This is the same case as the failure of the simple version of the decision tree algorithm.
3. We run out of examples and have several hypotheses remaining in the version space. This means the version space represents a disjunction of hypotheses. For any new example, if all the disjuncts agree, then we can return their classification of the example. If they disagree, one possibility is to take the majority vote.

We leave as an exercise the application of the VERSION-SPACE-LEARNING algorithm to the restaurant data.

There are two principal drawbacks to the version-space approach:

- If the domain contains noise or insufficient attributes for exact classification, the version space will always collapse.
- If we allow unlimited disjunction in the hypothesis space, the S-set will always contain a single most-specific hypothesis, namely, the disjunction of the descriptions of the positive examples seen to date. Similarly, the G-set will contain just the negation of the disjunction of the descriptions of the negative examples.
- For some hypothesis spaces, the number of elements in the S-set or G-set may grow exponentially in the number of attributes, even though efficient learning algorithms exist for those hypothesis spaces.

GENERALIZATION HIERARCHY

To date, no completely successful solution has been found for the problem of noise. The problem of disjunction can be addressed by allowing only limited forms of disjunction or by including a **generalization hierarchy** of more general predicates. For example, instead of using the disjunction $WaitEstimate(x, \mathit{30\text{-}60}) \lor WaitEstimate(x, >60)$, we might use the single literal $LongWait(x)$. The set of generalization and specialization operations can be easily extended to handle this.

The pure version space algorithm was first applied in the Meta-DENDRAL system, which was designed to learn rules for predicting how molecules would break into pieces in a mass spectrometer (Buchanan and Mitchell, 1978). Meta-DENDRAL was able to generate rules that were sufficiently novel to warrant publication in a journal of analytical chemistry—the first real scientific knowledge generated by a computer program. It was also used in the elegant LEX system (Mitchell *et al.*, 1983), which was able to learn to solve symbolic integration problems by studying its own successes and failures. Although version space methods are probably not practical in most real-world learning problems, mainly because of noise, they provide a good deal of insight into the logical structure of hypothesis space.

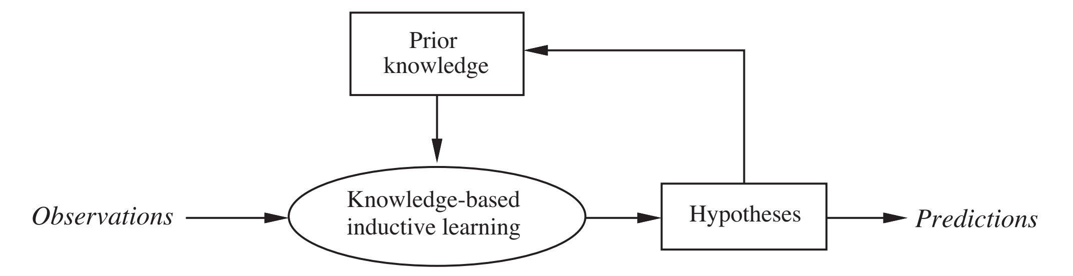


Figure 19.6 A cumulative learning process uses, and adds to, its stock of background knowledge over time.

19.2 KNOWLEDGE IN LEARNING

The preceding section described the simplest setting for inductive learning. To understand the role of prior knowledge, we need to talk about the logical relationships among hypotheses, example descriptions, and classifications. Let $\mathit{Descriptions}$ denote the conjunction of all the example descriptions in the training set, and let $\mathit{Classifications}$ denote the conjunction of all the example classifications. Then a $\mathit{Hypothesis}$ that "explains the observations" must satisfy the following property (recall that $\models$ means "logically entails"):

$$\mathit{Hypothesis} \land \mathit{Descriptions} \models \mathit{Classifications}\ . \tag{19.3}$$

ENTAILMENT CONSTRAINT

We call this kind of relationship an **entailment constraint**, in which $\mathit{Hypothesis}$ is the "unknown." Pure inductive learning means solving this constraint, where $\mathit{Hypothesis}$ is drawn from some predefined hypothesis space. For example, if we consider a decision tree as a logical formula (see Equation (19.1) on page 769), then a decision tree that is consistent with all the examples will satisfy Equation (19.3). If we place *no* restrictions on the logical form of the hypothesis, of course, then $\mathit{Hypothesis} = \mathit{Classifications}$ also satisfies the constraint. Ockham's razor tells us to prefer *small*, consistent hypotheses, so we try to do better than simply memorizing the examples.

This simple knowledge-free picture of inductive learning persisted until the early 1980s. The modern approach is to design agents that *already know something* and are trying to learn some more. This may not sound like a terrifically deep insight, but it makes quite a difference to the way we design agents. It might also have some relevance to our theories about how science itself works. The general idea is shown schematically in Figure 19.6.

An autonomous learning agent that uses background knowledge must somehow obtain the background knowledge in the first place, in order for it to be used in the new learning episodes. This method must itself be a learning process. The agent's life history will therefore be characterized by *cumulative*, or *incremental*, development. Presumably, the agent could start out with nothing, performing inductions *in vacuo* like a good little pure induction program. But once it has eaten from the Tree of Knowledge, it can no longer pursue such naive speculations and should use its background knowledge to learn more and more effectively. The question is then how to actually do this.

19.2.1 Some simple examples

Let us consider some commonsense examples of learning with background knowledge. Many apparently rational cases of inferential behavior in the face of observations clearly do not follow the simple principles of pure induction.

- Sometimes one leaps to general conclusions after only one observation. Gary Larson once drew a cartoon in which a bespectacled caveman, Zog, is roasting his lizard on the end of a pointed stick. He is watched by an amazed crowd of his less intellectual contemporaries, who have been using their bare hands to hold their victuals over the fire. This enlightening experience is enough to convince the watchers of a general principle of painless cooking.
- Or consider the case of the traveler to Brazil meeting her first Brazilian. On hearing him speak Portuguese, she immediately concludes that Brazilians speak Portuguese, yet on discovering that his name is Fernando, she does not conclude that all Brazilians are called Fernando. Similar examples appear in science. For example, when a freshman physics student measures the density and conductance of a sample of copper at a particular temperature, she is quite confident in generalizing those values to all pieces of copper. Yet when she measures its mass, she does not even consider the hypothesis that all pieces of copper have that mass. On the other hand, it would be quite reasonable to make such a generalization over all pennies.
- Finally, consider the case of a pharmacologically ignorant but diagnostically sophisticated medical student observing a consulting session between a patient and an expert internist. After a series of questions and answers, the expert tells the patient to take a course of a particular antibiotic. The medical student infers the general rule that that particular antibiotic is effective for a particular type of infection.

These are all cases in which *the use of background knowledge allows much faster learning than one might expect from a pure induction program.*

19.2.2 Some general schemes

In each of the preceding examples, one can appeal to prior knowledge to try to justify the generalizations chosen. We will now look at what kinds of entailment constraints are operating in each case. The constraints will involve the $\mathit{Background}$ knowledge, in addition to the $\mathit{Hypothesis}$ and the observed $\mathit{Descriptions}$ and $\mathit{Classifications}$.

EXPLANATION-BASED LEARNING

In the case of lizard toasting, the cavemen generalize by *explaining* the success of the pointed stick: it supports the lizard while keeping the hand away from the fire. From this explanation, they can infer a general rule: that any long, rigid, sharp object can be used to toast small, soft-bodied edibles. This kind of generalization process has been called **explanation-based learning**, or **EBL**. Notice that the general rule *follows logically* from the background knowledge possessed by the cavemen. Hence, the entailment constraints satisfied by EBL are the following:

$$\mathit{Hypothesis} \wedge \mathit{Descriptions} \models \mathit{Classifications}$$
$$\mathit{Background} \models \mathit{Hypothesis}\ .$$

Because EBL uses Equation (19.3), it was initially thought to be a way to learn from examples. But because it requires that the background knowledge be sufficient to explain the $Hypothesis$, which in turn explains the observations, *the agent does not actually learn anything factually new from the example.* The agent *could have* derived the example from what it already knew, although that might have required an unreasonable amount of computation. EBL is now viewed as a method for converting first-principles theories into useful, special-purpose knowledge. We describe algorithms for EBL in Section 19.3.

The situation of our traveler in Brazil is quite different, for she cannot necessarily explain why Fernando speaks the way he does, unless she knows her papal bulls. Moreover, the same generalization would be forthcoming from a traveler entirely ignorant of colonial history. The relevant prior knowledge in this case is that, within any given country, most people tend to speak the same language; on the other hand, Fernando is not assumed to be the name of all Brazilians because this kind of regularity does not hold for names. Similarly, the freshman physics student also would be hard put to explain the particular values that she discovers for the conductance and density of copper. She does know, however, that the material of which an object is composed and its temperature together determine its conductance.

RELEVANCE

In each case, the prior knowledge $Background$ concerns the **relevance** of a set of features to the goal predicate. This knowledge, *together with the observations*, allows the agent to infer a new, general rule that explains the observations:

$$\begin{aligned} &Hypothesis \wedge Descriptions \models Classifications \ , \\ &Background \wedge Descriptions \wedge Classifications \models Hypothesis \ . \end{aligned} \qquad (19.4)$$

RELEVANCE-BASED LEARNING

We call this kind of generalization **relevance-based learning**, or **RBL** (although the name is not standard). Notice that whereas RBL does make use of the content of the observations, it does not produce hypotheses that go beyond the logical content of the background knowledge and the observations. It is a *deductive* form of learning and cannot by itself account for the creation of new knowledge starting from scratch.

In the case of the medical student watching the expert, we assume that the student's prior knowledge is sufficient to infer the patient's disease D from the symptoms. This is not, however, enough to explain the fact that the doctor prescribes a particular medicine M. The student needs to propose another rule, namely, that M generally is effective against D. Given this rule and the student's prior knowledge, the student can now explain why the expert prescribes M in this particular case. We can generalize this example to come up with the entailment constraint

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications \ . \qquad (19.5)$$

That is, *the background knowledge and the new hypothesis combine to explain the examples.* As with pure inductive learning, the learning algorithm should propose hypotheses that are as simple as possible, consistent with this constraint. Algorithms that satisfy constraint (19.5) are called **knowledge-based inductive learning**, or **KBIL**, algorithms.

KNOWLEDGE-BASED INDUCTIVE LEARNING

INDUCTIVE LOGIC PROGRAMMING

KBIL algorithms, which are described in detail in Section 19.5, have been studied mainly in the field of **inductive logic programming**, or **ILP**. In ILP systems, prior knowledge plays two key roles in reducing the complexity of learning:

1. Because any hypothesis generated must be consistent with the prior knowledge as well as with the new observations, the effective hypothesis space size is reduced to include only those theories that are consistent with what is already known.
2. For any given set of observations, the size of the hypothesis required to construct an explanation for the observations can be much reduced, because the prior knowledge will be available to help out the new rules in explaining the observations. The smaller the hypothesis, the easier it is to find.

In addition to allowing the use of prior knowledge in induction, ILP systems can formulate hypotheses in general first-order logic, rather than in the restricted attribute-based language of Chapter 18. This means that they can learn in environments that cannot be understood by simpler systems.

19.3 EXPLANATION-BASED LEARNING

Explanation-based learning is a method for extracting general rules from individual observations. As an example, consider the problem of differentiating and simplifying algebraic expressions (Exercise 9.17). If we differentiate an expression such as X^2 with respect to X, we obtain $2X$. (We use a capital letter for the arithmetic unknown X, to distinguish it from the logical variable x.) In a logical reasoning system, the goal might be expressed as $\text{ASK}(\mathit{Derivative}(X^2, X) = d,\ \mathit{KB})$, with solution $d = 2X$.

Anyone who knows differential calculus can see this solution "by inspection" as a result of practice in solving such problems. A student encountering such problems for the first time, or a program with no experience, will have a much more difficult job. Application of the standard rules of differentiation eventually yields the expression $1 \times (2 \times (X^{(2-1)}))$, and eventually this simplifies to $2X$. In the authors' logic programming implementation, this takes 136 proof steps, of which 99 are on dead-end branches in the proof. After such an experience, we would like the program to solve the same problem much more quickly the next time it arises.

MEMOIZATION

The technique of **memoization** has long been used in computer science to speed up programs by saving the results of computation. The basic idea of memo functions is to accumulate a database of input–output pairs; when the function is called, it first checks the database to see whether it can avoid solving the problem from scratch. Explanation-based learning takes this a good deal further, by creating *general* rules that cover an entire class of cases. In the case of differentiation, memoization would remember that the derivative of X^2 with respect to X is $2X$, but would leave the agent to calculate the derivative of Z^2 with respect to Z from scratch. We would like to be able to extract the general rule that for any arithmetic unknown u, the derivative of u^2 with respect to u is $2u$. (An even more general rule for u^n can also be produced, but the current example suffices to make the point.) In logical terms, this is expressed by the rule

$$\mathit{ArithmeticUnknown}(u) \Rightarrow \mathit{Derivative}(u^2, u) = 2u\ .$$

If the knowledge base contains such a rule, then any new case that is an instance of this rule can be solved immediately.

This is, of course, merely a trivial example of a very general phenomenon. Once something is understood, it can be generalized and reused in other circumstances. It becomes an "obvious" step and can then be used as a building block in solving problems still more complex. Alfred North Whitehead (1911), co-author with Bertrand Russell of *Principia Mathematica*, wrote *"Civilization advances by extending the number of important operations that we can do without thinking about them,"* perhaps himself applying EBL to his understanding of events such as Zog's discovery. If you have understood the basic idea of the differentiation example, then your brain is already busily trying to extract the general principles of explanation-based learning from it. Notice that you hadn't *already* invented EBL before you saw the example. Like the cavemen watching Zog, you (and we) needed an example before we could generate the basic principles. This is because *explaining why* something is a good idea is much easier than coming up with the idea in the first place.

19.3.1 Extracting general rules from examples

The basic idea behind EBL is first to construct an explanation of the observation using prior knowledge, and then to establish a definition of the class of cases for which the same explanation structure can be used. This definition provides the basis for a rule covering all of the cases in the class. The "explanation" can be a logical proof, but more generally it can be any reasoning or problem-solving process whose steps are well defined. The key is to be able to identify the necessary conditions for those same steps to apply to another case.

We will use for our reasoning system the simple backward-chaining theorem prover described in Chapter 9. The proof tree for $Derivative(X^2, X) = 2X$ is too large to use as an example, so we will use a simpler problem to illustrate the generalization method. Suppose our problem is to simplify $1 \times (0 + X)$. The knowledge base includes the following rules:

$$\begin{array}{l}
Rewrite(u, v) \land Simplify(v, w) \;\Rightarrow\; Simplify(u, w)\;. \\
Primitive(u) \;\Rightarrow\; Simplify(u, u)\;. \\
ArithmeticUnknown(u) \;\Rightarrow\; Primitive(u)\;. \\
Number(u) \;\Rightarrow\; Primitive(u)\;. \\
Rewrite(1 \times u, u)\;. \\
Rewrite(0 + u, u)\;. \\
\qquad\vdots
\end{array}$$

The proof that the answer is X is shown in the top half of Figure 19.7. The EBL method actually constructs two proof trees simultaneously. The second proof tree uses a *variabilized* goal in which the constants from the original goal are replaced by variables. As the original proof proceeds, the variabilized proof proceeds in step, using *exactly the same rule applications*. This could cause some of the variables to become instantiated. For example, in order to use the rule $Rewrite(1 \times u, u)$, the variable x in the subgoal $Rewrite(x \times (y + z), v)$ must be bound to 1. Similarly, y must be bound to 0 in the subgoal $Rewrite(y + z, v')$ in order to use the rule $Rewrite(0 + u, u)$. Once we have the generalized proof tree, we take the leaves

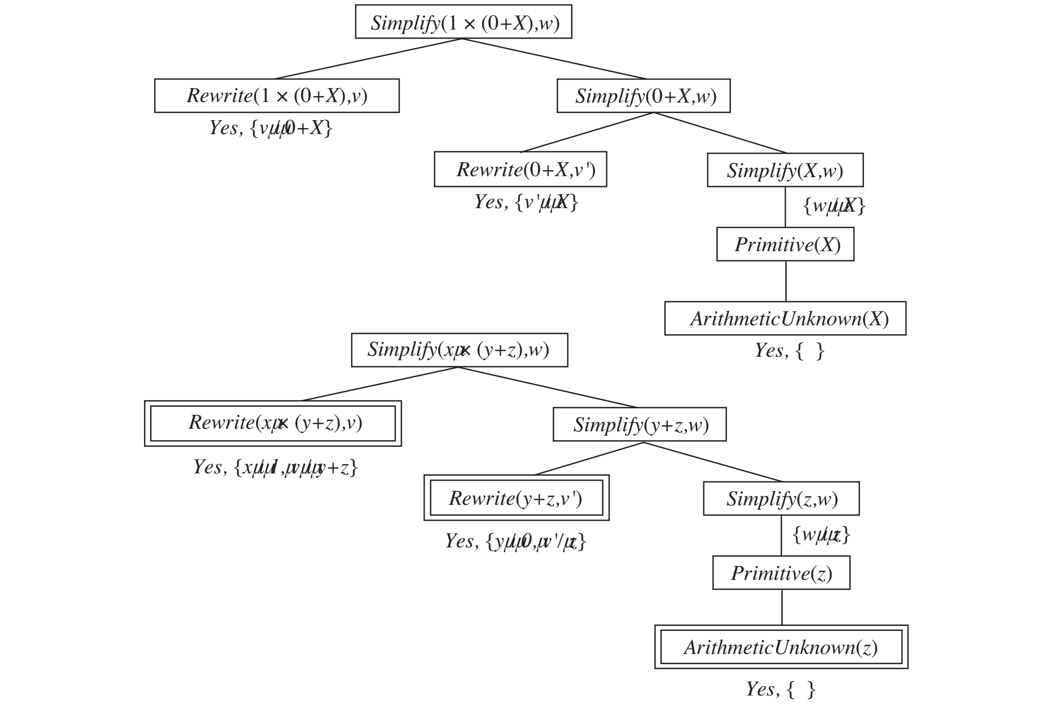


Figure 19.7 Proof trees for the simplification problem. The first tree shows the proof for the original problem instance, from which we can derive

$$ArithmeticUnknown(z) \Rightarrow Simplify(1 \times (0+z), z) .$$

The second tree shows the proof for a problem instance with all constants replaced by variables, from which we can derive a variety of other rules.

(with the necessary bindings) and form a general rule for the goal predicate:

$$Rewrite(1 \times (0+z), 0+z) \land Rewrite(0+z, z) \land ArithmeticUnknown(z)$$
$$\Rightarrow Simplify(1 \times (0+z), z) .$$

Notice that the first two conditions on the left-hand side are true *regardless of the value of* z. We can therefore drop them from the rule, yielding

$$ArithmeticUnknown(z) \Rightarrow Simplify(1 \times (0+z), z) .$$

In general, conditions can be dropped from the final rule if they impose no constraints on the variables on the right-hand side of the rule, because the resulting rule will still be true and will be more efficient. Notice that we cannot drop the condition $ArithmeticUnknown(z)$, because not all possible values of z are arithmetic unknowns. Values other than arithmetic unknowns might require different forms of simplification: for example, if z were 2×3, then the correct simplification of $1 \times (0 + (2 \times 3))$ would be 6 and not 2×3.

To recap, the basic EBL process works as follows:

1. Given an example, construct a proof that the goal predicate applies to the example using the available background knowledge.

2. In parallel, construct a generalized proof tree for the variabilized goal using the same inference steps as in the original proof.
3. Construct a new rule whose left-hand side consists of the leaves of the proof tree and whose right-hand side is the variabilized goal (after applying the necessary bindings from the generalized proof).
4. Drop any conditions from the left-hand side that are true regardless of the values of the variables in the goal.

19.3.2 Improving efficiency

The generalized proof tree in Figure 19.7 actually yields more than one generalized rule. For example, if we terminate, or **prune**, the growth of the right-hand branch in the proof tree when it reaches the $Primitive$ step, we get the rule

$$Primitive(z) \Rightarrow Simplify(1 \times (0 + z), z) .$$

This rule is as valid as, but *more general* than, the rule using $ArithmeticUnknown$, because it covers cases where z is a number. We can extract a still more general rule by pruning after the step $Simplify(y + z, w)$, yielding the rule

$$Simplify(y + z, w) \Rightarrow Simplify(1 \times (y + z), w) .$$

In general, a rule can be extracted from *any partial subtree* of the generalized proof tree. Now we have a problem: which of these rules do we choose?

The choice of which rule to generate comes down to the question of efficiency. There are three factors involved in the analysis of efficiency gains from EBL:

1. Adding large numbers of rules can slow down the reasoning process, because the inference mechanism must still check those rules even in cases where they do not yield a solution. In other words, it increases the **branching factor** in the search space.
2. To compensate for the slowdown in reasoning, the derived rules must offer significant increases in speed for the cases that they do cover. These increases come about mainly because the derived rules avoid dead ends that would otherwise be taken, but also because they shorten the proof itself.
3. Derived rules should be as general as possible, so that they apply to the largest possible set of cases.

OPERATIONALITY

A common approach to ensuring that derived rules are efficient is to insist on the **operationality** of each subgoal in the rule. A subgoal is operational if it is "easy" to solve. For example, the subgoal $Primitive(z)$ is easy to solve, requiring at most two steps, whereas the subgoal $Simplify(y + z, w)$ could lead to an arbitrary amount of inference, depending on the values of y and z. If a test for operationality is carried out at each step in the construction of the generalized proof, then we can prune the rest of a branch as soon as an operational subgoal is found, keeping just the operational subgoal as a conjunct of the new rule.

Unfortunately, there is usually a tradeoff between operationality and generality. More specific subgoals are generally easier to solve but cover fewer cases. Also, operationality is a matter of degree: one or two steps is definitely operational, but what about 10 or 100?

Finally, the cost of solving a given subgoal depends on what other rules are available in the knowledge base. It can go up or down as more rules are added. Thus, EBL systems really face a very complex optimization problem in trying to maximize the efficiency of a given initial knowledge base. It is sometimes possible to derive a mathematical model of the effect on overall efficiency of adding a given rule and to use this model to select the best rule to add. The analysis can become very complicated, however, especially when recursive rules are involved. One promising approach is to address the problem of efficiency empirically, simply by adding several rules and seeing which ones are useful and actually speed things up.

Empirical analysis of efficiency is actually at the heart of EBL. What we have been calling loosely the "efficiency of a given knowledge base" is actually the average-case complexity on a distribution of problems. *By generalizing from past example problems, EBL makes the knowledge base more efficient for the kind of problems that it is reasonable to expect.* This works as long as the distribution of past examples is roughly the same as for future examples—the same assumption used for PAC-learning in Section 18.5. If the EBL system is carefully engineered, it is possible to obtain significant speedups. For example, a very large Prolog-based natural language system designed for speech-to-speech translation between Swedish and English was able to achieve real-time performance only by the application of EBL to the parsing process (Samuelsson and Rayner, 1991).

19.4 Learning Using Relevance Information

Our traveler in Brazil seems to be able to make a confident generalization concerning the language spoken by other Brazilians. The inference is sanctioned by her background knowledge, namely, that people in a given country (usually) speak the same language. We can express this in first-order logic as follows:[2]

$$\mathit{Nationality}(x,n) \wedge \mathit{Nationality}(y,n) \wedge \mathit{Language}(x,l) \;\Rightarrow\; \mathit{Language}(y,l)\ . \quad (19.6)$$

(Literal translation: "If x and y have the same nationality n and x speaks language l, then y also speaks it.") It is not difficult to show that, from this sentence and the observation that

$$\mathit{Nationality}(\mathit{Fernando}, \mathit{Brazil}) \wedge \mathit{Language}(\mathit{Fernando}, \mathit{Portuguese})\ ,$$

the following conclusion is entailed (see Exercise 19.1):

$$\mathit{Nationality}(x, \mathit{Brazil}) \;\Rightarrow\; \mathit{Language}(x, \mathit{Portuguese})\ .$$

Sentences such as (19.6) express a strict form of relevance: given nationality, language is fully determined. (Put another way: language is a function of nationality.) These sentences are called **functional dependencies** or **determinations**. They occur so commonly in certain kinds of applications (e.g., defining database designs) that a special syntax is used to write them. We adopt the notation of Davies (1985):

FUNCTIONAL DEPENDENCY

DETERMINATION

$$\mathit{Nationality}(x,n) \succ \mathit{Language}(x,l)\ .$$

[2] We assume for the sake of simplicity that a person speaks only one language. Clearly, the rule would have to be amended for countries such as Switzerland and India.

As usual, this is simply a syntactic sugaring, but it makes it clear that the determination is really a relationship between the predicates: nationality determines language. The relevant properties determining conductance and density can be expressed similarly:

$$\begin{aligned}&\mathit{Material}(x,m) \wedge \mathit{Temperature}(x,t) \succ \mathit{Conductance}(x,\rho)\ ;\\&\mathit{Material}(x,m) \wedge \mathit{Temperature}(x,t) \succ \mathit{Density}(x,d)\ .\end{aligned}$$

The corresponding generalizations follow logically from the determinations and observations.

19.4.1 Determining the hypothesis space

Although the determinations sanction general conclusions concerning all Brazilians, or all pieces of copper at a given temperature, they cannot, of course, yield a general predictive theory for *all* nationalities, or for *all* temperatures and materials, from a single example. Their main effect can be seen as limiting the space of hypotheses that the learning agent need consider. In predicting conductance, for example, one need consider only material and temperature and can ignore mass, ownership, day of the week, the current president, and so on. Hypotheses can certainly include terms that are in turn determined by material and temperature, such as molecular structure, thermal energy, or free-electron density. *Determinations specify a sufficient basis vocabulary from which to construct hypotheses concerning the target predicate.* This statement can be proven by showing that a given determination is logically equivalent to a statement that the correct definition of the target predicate is one of the set of all definitions expressible using the predicates on the left-hand side of the determination.

Intuitively, it is clear that a reduction in the hypothesis space size should make it easier to learn the target predicate. Using the basic results of computational learning theory (Section 18.5), we can quantify the possible gains. First, recall that for Boolean functions, $\log(|\mathcal{H}|)$ examples are required to converge to a reasonable hypothesis, where $|\mathcal{H}|$ is the size of the hypothesis space. If the learner has n Boolean features with which to construct hypotheses, then, in the absence of further restrictions, $|\mathcal{H}| = O(2^{2^n})$, so the number of examples is $O(2^n)$. If the determination contains d predicates in the left-hand side, the learner will require only $O(2^d)$ examples, a reduction of $O(2^{n-d})$.

19.4.2 Learning and using relevance information

As we stated in the introduction to this chapter, prior knowledge is useful in learning; but it too has to be learned. In order to provide a complete story of relevance-based learning, we must therefore provide a learning algorithm for determinations. The learning algorithm we now present is based on a straightforward attempt to find the simplest determination consistent with the observations. A determination $P \succ Q$ says that if any examples match on P, then they must also match on Q. A determination is therefore consistent with a set of examples if every pair that matches on the predicates on the left-hand side also matches on the goal predicate. For example, suppose we have the following examples of conductance measurements on material samples:

```
function MINIMAL-CONSISTENT-DET($E, A$) returns a set of attributes
   inputs: $E$, a set of examples
           $A$, a set of attributes, of size $n$

   for $i$ = 0 to $n$ do
      for each subset $A_i$ of $A$ of size $i$ do
         if CONSISTENT-DET?($A_i, E$) then return $A_i$
------------------------------------------------------------------------
function CONSISTENT-DET?($A, E$) returns a truth value
   inputs: $A$, a set of attributes
           $E$, a set of examples
   local variables: $H$, a hash table

   for each example $e$ in $E$ do
      if some example in $H$ has the same values as $e$ for the attributes $A$
         but a different classification then return $false$
      store the class of $e$ in $H$, indexed by the values for attributes $A$ of the example $e$
   return $true$
```

Figure 19.8 An algorithm for finding a minimal consistent determination.

Sample	Mass	Temperature	Material	Size	Conductance
S1	12	26	Copper	3	0.59
S1	12	100	Copper	3	0.57
S2	24	26	Copper	6	0.59
S3	12	26	Lead	2	0.05
S3	12	100	Lead	2	0.04
S4	24	26	Lead	4	0.05

The minimal consistent determination is $Material \wedge Temperature \succ Conductance$. There is a nonminimal but consistent determination, namely, $Mass \wedge Size \wedge Temperature \succ Conductance$. This is consistent with the examples because mass and size determine density and, in our data set, we do not have two different materials with the same density. As usual, we would need a larger sample set in order to eliminate a nearly correct hypothesis.

There are several possible algorithms for finding minimal consistent determinations. The most obvious approach is to conduct a search through the space of determinations, checking all determinations with one predicate, two predicates, and so on, until a consistent determination is found. We will assume a simple attribute-based representation, like that used for decision tree learning in Chapter 18. A determination d will be represented by the set of attributes on the left-hand side, because the target predicate is assumed to be fixed. The basic algorithm is outlined in Figure 19.8.

The time complexity of this algorithm depends on the size of the smallest consistent determination. Suppose this determination has p attributes out of the n total attributes. Then the algorithm will not find it until searching the subsets of A of size p. There are $\binom{n}{p} = O(n^p)$

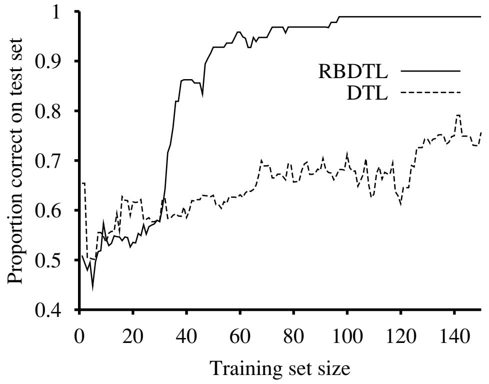


Figure 19.9 A performance comparison between DECISION-TREE-LEARNING and RBDTL on randomly generated data for a target function that depends on only 5 of 16 attributes.

such subsets; hence the algorithm is exponential in the size of the minimal determination. It turns out that the problem is NP-complete, so we cannot expect to do better in the general case. In most domains, however, there will be sufficient local structure (see Chapter 14 for a definition of locally structured domains) that p will be small.

Given an algorithm for learning determinations, a learning agent has a way to construct a minimal hypothesis within which to learn the target predicate. For example, we can combine MINIMAL-CONSISTENT-DET with the DECISION-TREE-LEARNING algorithm. This yields a relevance-based decision-tree learning algorithm RBDTL that first identifies a minimal set of relevant attributes and then passes this set to the decision tree algorithm for learning. Unlike DECISION-TREE-LEARNING, RBDTL simultaneously learns and uses relevance information in order to minimize its hypothesis space. We expect that RBDTL will learn faster than DECISION-TREE-LEARNING, and this is in fact the case. Figure 19.9 shows the learning performance for the two algorithms on randomly generated data for a function that depends on only 5 of 16 attributes. Obviously, in cases where all the available attributes are relevant, RBDTL will show no advantage.

DECLARATIVE BIAS

This section has only scratched the surface of the field of **declarative bias**, which aims to understand how prior knowledge can be used to identify the appropriate hypothesis space within which to search for the correct target definition. There are many unanswered questions:

- How can the algorithms be extended to handle noise?
- Can we handle continuous-valued variables?
- How can other kinds of prior knowledge be used, besides determinations?
- How can the algorithms be generalized to cover any first-order theory, rather than just an attribute-based representation?

Some of these questions are addressed in the next section.

19.5 Inductive Logic Programming

Inductive logic programming (ILP) combines inductive methods with the power of first-order representations, concentrating in particular on the representation of hypotheses as logic programs.[3] It has gained popularity for three reasons. First, ILP offers a rigorous approach to the general knowledge-based inductive learning problem. Second, it offers complete algorithms for inducing general, first-order theories from examples, which can therefore learn successfully in domains where attribute-based algorithms are hard to apply. An example is in learning how protein structures fold (Figure 19.10). The three-dimensional configuration of a protein molecule cannot be represented reasonably by a set of attributes, because the configuration inherently refers to *relationships* between objects, not to attributes of a single object. First-order logic is an appropriate language for describing the relationships. Third, inductive logic programming produces hypotheses that are (relatively) easy for humans to read. For example, the English translation in Figure 19.10 can be scrutinized and criticized by working biologists. This means that inductive logic programming systems can participate in the scientific cycle of experimentation, hypothesis generation, debate, and refutation. Such participation would not be possible for systems that generate “black-box” classifiers, such as neural networks.

19.5.1 An example

Recall from Equation (19.5) that the general knowledge-based induction problem is to “solve” the entailment constraint

$$Background \land Hypothesis \land Descriptions \models Classifications$$

for the unknown $Hypothesis$, given the $Background$ knowledge and examples described by $Descriptions$ and $Classifications$. To illustrate this, we will use the problem of learning family relationships from examples. The descriptions will consist of an extended family tree, described in terms of $Mother$, $Father$, and $Married$ relations and $Male$ and $Female$ properties. As an example, we will use the family tree from Exercise 8.14, shown here in Figure 19.11. The corresponding descriptions are as follows:

$$\begin{array}{lll}
Father(Philip, Charles) & Father(Philip, Anne) & \ldots \\
Mother(Mum, Margaret) & Mother(Mum, Elizabeth) & \ldots \\
Married(Diana, Charles) & Married(Elizabeth, Philip) & \ldots \\
Male(Philip) & Male(Charles) & \ldots \\
Female(Beatrice) & Female(Margaret) & \ldots
\end{array}$$

The sentences in $Classifications$ depend on the target concept being learned. We might want to learn $Grandparent$, $BrotherInLaw$, or $Ancestor$, for example. For $Grandparent$, the

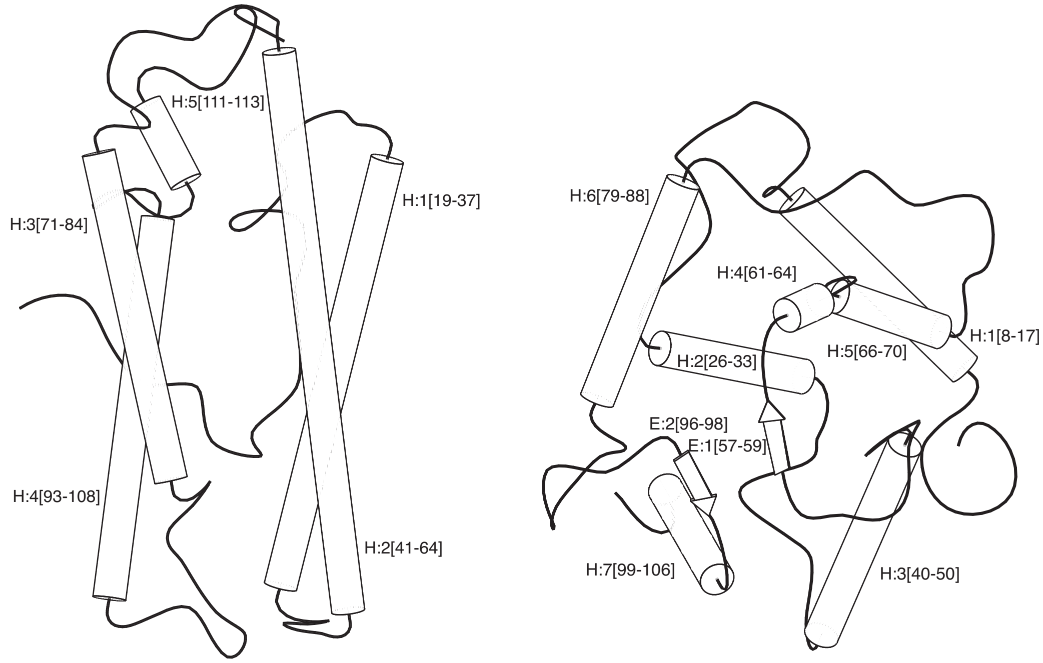
[3] It might be appropriate at this point for the reader to refer to Chapter 7 for some of the underlying concepts, including Horn clauses, conjunctive normal form, unification, and resolution.

complete set of *Classifications* contains $20 \times 20 = 400$ conjuncts of the form

$$\begin{array}{lll} Grandparent(Mum, Charles) & Grandparent(Elizabeth, Beatrice) & \ldots \\ \neg Grandparent(Mum, Harry) & \neg Grandparent(Spencer, Peter) & \ldots \end{array}$$

We could of course learn from a subset of this complete set.

The object of an inductive learning program is to come up with a set of sentences for the *Hypothesis* such that the entailment constraint is satisfied. Suppose, for the moment, that the agent has no background knowledge: *Background* is empty. Then one possible solution



2mhr - Four-helical up-and-down bundle

(a)

1omd - EF-Hand

(b)

Figure 19.10 (a) and (b) show positive and negative examples, respectively, of the "four-helical up-and-down bundle" concept in the domain of protein folding. Each example structure is coded into a logical expression of about 100 conjuncts such as $TotalLength(D2mhr, 118) \wedge NumberHelices(D2mhr, 6) \wedge \ldots$. From these descriptions and from classifications such as $Fold(\textsc{Four-Helical-Up-And-Down-Bundle}, D2mhr)$, the ILP system PROGOL (Muggleton, 1995) learned the following rule:

$$\begin{array}{l} Fold(\textsc{Four-Helical-Up-And-Down-Bundle}, p) \Leftarrow \\ \quad Helix(p, h_1) \wedge Length(h_1, \textsc{High}) \wedge Position(p, h_1, n) \\ \quad \wedge\ (1 \leq n \leq 3) \wedge Adjacent(p, h_1, h_2) \wedge Helix(p, h_2)\ . \end{array}$$

This kind of rule could not be learned, or even represented, by an attribute-based mechanism such as we saw in previous chapters. The rule can be translated into English as " Protein p has fold class "Four-helical up-and-down-bundle" if it contains a long helix h_1 at a secondary structure position between 1 and 3 and h_1 is next to a second helix."

for *Hypothesis* is the following:

$$\begin{aligned} Grandparent(x,y) \;\; \Leftrightarrow \;\; & [\exists\, z \;\; Mother(x,z) \land Mother(z,y)] \\ \lor \;\; & [\exists\, z \;\; Mother(x,z) \land Father(z,y)] \\ \lor \;\; & [\exists\, z \;\; Father(x,z) \land Mother(z,y)] \\ \lor \;\; & [\exists\, z \;\; Father(x,z) \land Father(z,y)] \; . \end{aligned}$$

Notice that an attribute-based learning algorithm, such as DECISION-TREE-LEARNING, will get nowhere in solving this problem. In order to express *Grandparent* as an attribute (i.e., a unary predicate), we would need to make *pairs* of people into objects:

$$Grandparent(\langle Mum, Charles\rangle) \ldots$$

Then we get stuck in trying to represent the example descriptions. The only possible attributes are horrible things such as

$$FirstElementIsMotherOfElizabeth(\langle Mum, Charles\rangle) \; .$$

The definition of *Grandparent* in terms of these attributes simply becomes a large disjunction of specific cases that does not generalize to new examples at all. *Attribute-based learning algorithms are incapable of learning relational predicates.* Thus, one of the principal advantages of ILP algorithms is their applicability to a much wider range of problems, including relational problems.

The reader will certainly have noticed that a little bit of background knowledge would help in the representation of the *Grandparent* definition. For example, if *Background* included the sentence

$$Parent(x,y) \;\; \Leftrightarrow \;\; [Mother(x,y) \lor Father(x,y)] \; ,$$

then the definition of *Grandparent* would be reduced to

$$Grandparent(x,y) \;\; \Leftrightarrow \;\; [\exists\, z \;\; Parent(x,z) \land Parent(z,y)] \; .$$

This shows how background knowledge can dramatically reduce the size of hypotheses required to explain the observations.

It is also possible for ILP algorithms to *create* new predicates in order to facilitate the expression of explanatory hypotheses. Given the example data shown earlier, it is entirely reasonable for the ILP program to propose an additional predicate, which we would call

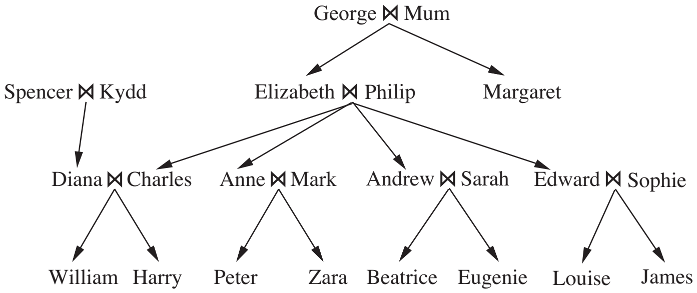


Figure 19.11 A typical family tree.

CONSTRUCTIVE INDUCTION

"*Parent*," in order to simplify the definitions of the target predicates. Algorithms that can generate new predicates are called **constructive induction** algorithms. Clearly, constructive induction is a necessary part of the picture of cumulative learning. It has been one of the hardest problems in machine learning, but some ILP techniques provide effective mechanisms for achieving it.

In the rest of this chapter, we will study the two principal approaches to ILP. The first uses a generalization of decision tree methods, and the second uses techniques based on inverting a resolution proof.

19.5.2 Top-down inductive learning methods

The first approach to ILP works by starting with a very general rule and gradually specializing it so that it fits the data. This is essentially what happens in decision-tree learning, where a decision tree is gradually grown until it is consistent with the observations. To do ILP we use first-order literals instead of attributes, and the hypothesis is a set of clauses instead of a decision tree. This section describes FOIL (Quinlan, 1990), one of the first ILP programs.

Suppose we are trying to learn a definition of the $Grandfather(x, y)$ predicate, using the same family data as before. As with decision-tree learning, we can divide the examples into positive and negative examples. Positive examples are

$$\langle George, Anne\rangle,\ \langle Philip, Peter\rangle,\ \langle Spencer, Harry\rangle,\ \ldots$$

and negative examples are

$$\langle George, Elizabeth\rangle,\ \langle Harry, Zara\rangle,\ \langle Charles, Philip\rangle,\ \ldots$$

Notice that each example is a *pair* of objects, because $Grandfather$ is a binary predicate. In all, there are 12 positive examples in the family tree and 388 negative examples (all the other pairs of people).

FOIL constructs a set of clauses, each with $Grandfather(x, y)$ as the head. The clauses must classify the 12 positive examples as instances of the $Grandfather(x, y)$ relationship, while ruling out the 388 negative examples. The clauses are Horn clauses, with the extension that negated literals are allowed in the body of a clause and are interpreted using negation as failure, as in Prolog. The initial clause has an empty body:

$$\Rightarrow\ Grandfather(x, y)\ .$$

This clause classifies every example as positive, so it needs to be specialized. We do this by adding literals one at a time to the left-hand side. Here are three potential additions:

$$\begin{array}{l} Father(x, y)\ \Rightarrow\ Grandfather(x, y)\ . \\ Parent(x, z)\ \Rightarrow\ Grandfather(x, y)\ . \\ Father(x, z)\ \Rightarrow\ Grandfather(x, y)\ . \end{array}$$

(Notice that we are assuming that a clause defining $Parent$ is already part of the background knowledge.) The first of these three clauses incorrectly classifies all of the 12 positive examples as negative and can thus be ignored. The second and third agree with all of the positive examples, but the second is incorrect on a larger fraction of the negative examples—twice as many, because it allows mothers as well as fathers. Hence, we prefer the third clause.

Now we need to specialize this clause further, to rule out the cases in which x is the father of some z, but z is not a parent of y. Adding the single literal $Parent(z, y)$ gives

$$Father(x, z) \land Parent(z, y) \ \Rightarrow \ Grandfather(x, y) \ ,$$

which correctly classifies all the examples. FOIL will find and choose this literal, thereby solving the learning task. In general, the solution is a set of Horn clauses, each of which implies the target predicate. For example, if we didn't have the $Parent$ predicate in our vocabulary, then the solution might be

$$Father(x, z) \land Father(z, y) \ \Rightarrow \ Grandfather(x, y)$$
$$Father(x, z) \land Mother(z, y) \ \Rightarrow \ Grandfather(x, y) \ .$$

Note that each of these clauses covers some of the positive examples, that together they cover all the positive examples, and that NEW-CLAUSE is designed in such a way that no clause will incorrectly cover a negative example. In general FOIL will have to search through many unsuccessful clauses before finding a correct solution.

This example is a very simple illustration of how FOIL operates. A sketch of the complete algorithm is shown in Figure 19.12. Essentially, the algorithm repeatedly constructs a clause, literal by literal, until it agrees with some subset of the positive examples and none of the negative examples. Then the positive examples covered by the clause are removed from the training set, and the process continues until no positive examples remain. The two main subroutines to be explained are NEW-LITERALS, which constructs all possible new literals to add to the clause, and CHOOSE-LITERAL, which selects a literal to add.

NEW-LITERALS takes a clause and constructs all possible "useful" literals that could be added to the clause. Let us use as an example the clause

$$Father(x, z) \ \Rightarrow \ Grandfather(x, y) \ .$$

There are three kinds of literals that can be added:

1. *Literals using predicates*: the literal can be negated or unnegated, any existing predicate (including the goal predicate) can be used, and the arguments must all be variables. Any variable can be used for any argument of the predicate, with one restriction: each literal must include *at least one* variable from an earlier literal or from the head of the clause. Literals such as $Mother(z, u)$, $Married(z, z)$, $\neg Male(y)$, and $Grandfather(v, x)$ are allowed, whereas $Married(u, v)$ is not. Notice that the use of the predicate from the head of the clause allows FOIL to learn *recursive* definitions.
2. *Equality and inequality literals*: these relate variables already appearing in the clause. For example, we might add $z \neq x$. These literals can also include user-specified constants. For learning arithmetic we might use 0 and 1, and for learning list functions we might use the empty list [ ].
3. *Arithmetic comparisons*: when dealing with functions of continuous variables, literals such as $x > y$ and $y \leq z$ can be added. As in decision-tree learning, a constant threshold value can be chosen to maximize the discriminatory power of the test.

The resulting branching factor in this search space is very large (see Exercise 19.6), but FOIL can also use type information to reduce it. For example, if the domain included numbers as

```
function FOIL(examples, target) returns a set of Horn clauses
  inputs: examples, set of examples
          target, a literal for the goal predicate
  local variables: clauses, set of clauses, initially empty

  while examples contains positive examples do
     clause ← NEW-CLAUSE(examples, target)
     remove positive examples covered by clause from examples
     add clause to clauses
  return clauses
```

```
function NEW-CLAUSE(examples, target) returns a Horn clause
  local variables: clause, a clause with target as head and an empty body
                   l, a literal to be added to the clause
                   extended_examples, a set of examples with values for new variables

  extended_examples ← examples
  while extended_examples contains negative examples do
     l ← CHOOSE-LITERAL(NEW-LITERALS(clause), extended_examples)
     append l to the body of clause
     extended_examples ← set of examples created by applying EXTEND-EXAMPLE
        to each example in extended_examples
  return clause
```

```
function EXTEND-EXAMPLE(example, literal) returns a set of examples
  if example satisfies literal
     then return the set of examples created by extending example with
        each possible constant value for each new variable in literal
  else return the empty set
```

Figure 19.12 Sketch of the FOIL algorithm for learning sets of first-order Horn clauses from examples. NEW-LITERALS and CHOOSE-LITERAL are explained in the text.

well as people, type restrictions would prevent NEW-LITERALS from generating literals such as $Parent(x, n)$, where x is a person and n is a number.

CHOOSE-LITERAL uses a heuristic somewhat similar to information gain (see page 704) to decide which literal to add. The exact details are not important here, and a number of different variations have been tried. One interesting additional feature of FOIL is the use of Ockham's razor to eliminate some hypotheses. If a clause becomes longer (according to some metric) than the total length of the positive examples that the clause explains, that clause is not considered as a potential hypothesis. This technique provides a way to avoid overcomplex clauses that fit noise in the data.

FOIL and its relatives have been used to learn a wide variety of definitions. One of the most impressive demonstrations (Quinlan and Cameron-Jones, 1993) involved solving a long sequence of exercises on list-processing functions from Bratko's (1986) Prolog textbook. In each case, the program was able to learn a correct definition of the function from a small set of examples, using the previously learned functions as background knowledge.

19.5.3 Inductive learning with inverse deduction

INVERSE RESOLUTION

The second major approach to ILP involves inverting the normal deductive proof process. **Inverse resolution** is based on the observation that if the example $\mathit{Classifications}$ follow from $\mathit{Background} \wedge \mathit{Hypothesis} \wedge \mathit{Descriptions}$, then one must be able to prove this fact by resolution (because resolution is complete). If we can "run the proof backward," then we can find a $\mathit{Hypothesis}$ such that the proof goes through. The key, then, is to find a way to invert the resolution process.

We will show a backward proof process for inverse resolution that consists of individual backward steps. Recall that an ordinary resolution step takes two clauses C_1 and C_2 and resolves them to produce the **resolvent** C. An inverse resolution step takes a resolvent C and produces two clauses C_1 and C_2, such that C is the result of resolving C_1 and C_2. Alternatively, it may take a resolvent C and clause C_1 and produce a clause C_2 such that C is the result of resolving C_1 and C_2.

The early steps in an inverse resolution process are shown in Figure 19.13, where we focus on the positive example $\mathit{Grandparent}(\mathit{George}, \mathit{Anne})$. The process begins at the end of the proof (shown at the bottom of the figure). We take the resolvent C to be empty clause (i.e. a contradiction) and C_2 to be $\neg \mathit{Grandparent}(\mathit{George}, \mathit{Anne})$, which is the negation of the goal example. The first inverse step takes C and C_2 and generates the clause $\mathit{Grandparent}(\mathit{George}, \mathit{Anne})$ for C_1. The next step takes this clause as C and the clause $\mathit{Parent}(\mathit{Elizabeth}, \mathit{Anne})$ as C_2, and generates the clause

$$\neg \mathit{Parent}(\mathit{Elizabeth}, y) \vee \mathit{Grandparent}(\mathit{George}, y)$$

as C_1. The final step treats this clause as the resolvent. With $\mathit{Parent}(\mathit{George}, \mathit{Elizabeth})$ as C_2, one possible clause C_1 is the hypothesis

$$\mathit{Parent}(x, z) \wedge \mathit{Parent}(z, y) \;\Rightarrow\; \mathit{Grandparent}(x, y) .$$

Now we have a resolution proof that the hypothesis, descriptions, and background knowledge entail the classification $\mathit{Grandparent}(\mathit{George}, \mathit{Anne})$.

Clearly, inverse resolution involves a search. Each inverse resolution step is nondeterministic, because for any C, there can be many or even an infinite number of clauses C_1 and C_2 that resolve to C. For example, instead of choosing $\neg \mathit{Parent}(\mathit{Elizabeth}, y) \vee \mathit{Grandparent}(\mathit{George}, y)$ for C_1 in the last step of Figure 19.13, the inverse resolution step might have chosen any of the following sentences:

$$\begin{array}{l}
\neg \mathit{Parent}(\mathit{Elizabeth}, \mathit{Anne}) \vee \mathit{Grandparent}(\mathit{George}, \mathit{Anne}) . \\
\neg \mathit{Parent}(z, \mathit{Anne}) \vee \mathit{Grandparent}(\mathit{George}, \mathit{Anne}) . \\
\neg \mathit{Parent}(z, y) \vee \mathit{Grandparent}(\mathit{George}, y) . \\
\quad\vdots
\end{array}$$

(See Exercises 19.4 and 19.5.) Furthermore, the clauses that participate in each step can be chosen from the $\mathit{Background}$ knowledge, from the example $\mathit{Descriptions}$, from the negated

Classifications, or from hypothesized clauses that have already been generated in the inverse resolution tree. The large number of possibilities means a large branching factor (and therefore an inefficient search) without additional controls. A number of approaches to taming the search have been tried in implemented ILP systems:

1. Redundant choices can be eliminated—for example, by generating only the most specific hypotheses possible and by requiring that all the hypothesized clauses be consistent with each other, and with the observations. This last criterion would rule out the clause $\neg Parent(z,y) \lor Grandparent(George,y)$, listed before.
2. The proof strategy can be restricted. For example, we saw in Chapter 9 that **linear resolution** is a complete, restricted strategy. Linear resolution produces proof trees that have a linear branching structure—the whole tree follows one line, with only single clauses branching off that line (as in Figure 19.13).
3. The representation language can be restricted, for example by eliminating function symbols or by allowing only Horn clauses. For instance, PROGOL operates with Horn clauses using **inverse entailment**. The idea is to change the entailment constraint

   INVERSE ENTAILMENT

   $$Background \land Hypothesis \land Descriptions \models Classifications$$

   to the logically equivalent form

   $$Background \land Descriptions \land \neg Classifications \models \neg Hypothesis\,.$$

   From this, one can use a process similar to the normal Prolog Horn-clause deduction, with negation-as-failure to derive *Hypothesis*. Because it is restricted to Horn clauses, this is an incomplete method, but it can be more efficient than full resolution. It is also possible to apply complete inference with inverse entailment (Inoue, 2001).
4. Inference can be done with model checking rather than theorem proving. The PROGOL system (Muggleton, 1995) uses a form of model checking to limit the search. That

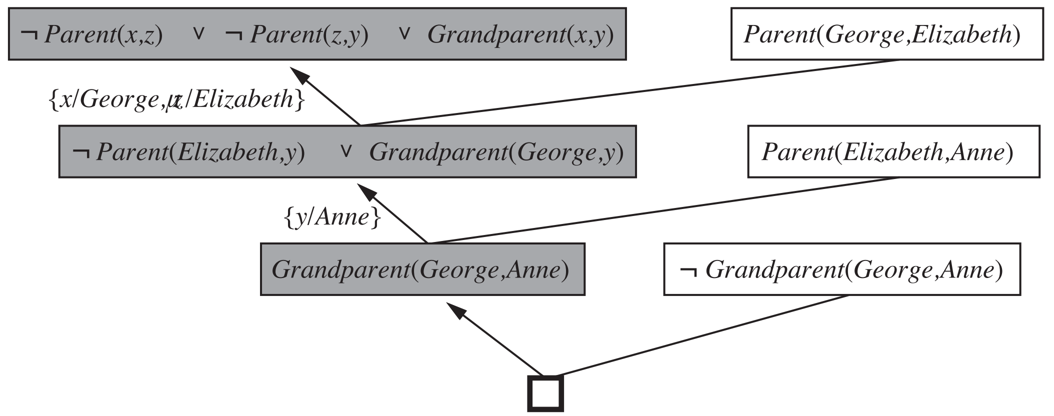

¬ Parent(x,z) ∨ ¬ Parent(z,y) ∨ Grandparent(x,y)
Parent(George,Elizabeth)
{x/George,z/Elizabeth}
¬ Parent(Elizabeth,y) ∨ Grandparent(George,y)
Parent(Elizabeth,Anne)
{y/Anne}
Grandparent(George,Anne)
¬ Grandparent(George,Anne)
□

Figure 19.13 Early steps in an inverse resolution process. The shaded clauses are generated by inverse resolution steps from the clause to the right and the clause below. The unshaded clauses are from the *Descriptions* and *Classifications* (including negated *Classifications*).

is, like answer set programming, it generates possible values for logical variables, and checks for consistency.

5. Inference can be done with ground propositional clauses rather than in first-order logic. The LINUS system (Lavra**u**c and D**u**zeroski, 1994) works by translating first-order theories into propositional logic, solving them with a propositional learning system, and then translating back. Working with propositional formulas can be more efficient on some problems, as we saw with SATPLAN in Chapter 10.

19.5.4 Making discoveries with inductive logic programming

An inverse resolution procedure that inverts a complete resolution strategy is, in principle, a complete algorithm for learning first-order theories. That is, if some unknown $\mathit{Hypothesis}$ generates a set of examples, then an inverse resolution procedure can generate $\mathit{Hypothesis}$ from the examples. This observation suggests an interesting possibility: Suppose that the available examples include a variety of trajectories of falling bodies. Would an inverse resolution program be theoretically capable of inferring the law of gravity? The answer is clearly yes, because the law of gravity allows one to explain the examples, given suitable background mathematics. Similarly, one can imagine that electromagnetism, quantum mechanics, and the theory of relativity are also within the scope of ILP programs. Of course, they are also within the scope of a monkey with a typewriter; we still need better heuristics and new ways to structure the search space.

One thing that inverse resolution systems *will* do for you is invent new predicates. This ability is often seen as somewhat magical, because computers are often thought of as "merely working with what they are given." In fact, new predicates fall directly out of the inverse resolution step. The simplest case arises in hypothesizing two new clauses C_1 and C_2, given a clause C. The resolution of C_1 and C_2 eliminates a literal that the two clauses share; hence, it is quite possible that the eliminated literal contained a predicate that does not appear in C. Thus, when working backward, one possibility is to generate a new predicate from which to reconstruct the missing literal.

Figure 19.14 shows an example in which the new predicate P is generated in the process of learning a definition for $Ancestor$. Once generated, P can be used in later inverse resolution steps. For example, a later step might hypothesize that $Mother(x, y) \Rightarrow P(x, y)$. Thus, the new predicate P has its meaning constrained by the generation of hypotheses that involve it. Another example might lead to the constraint $\mathit{Father}(x, y) \Rightarrow P(x, y)$. In other words, the predicate P is what we usually think of as the Parent relationship. As we mentioned earlier, the invention of new predicates can significantly reduce the size of the definition of the goal predicate. Hence, by including the ability to invent new predicates, inverse resolution systems can often solve learning problems that are infeasible with other techniques.

Some of the deepest revolutions in science come from the invention of new predicates and functions—for example, Galileo's invention of acceleration or Joule's invention of thermal energy. Once these terms are available, the discovery of new laws becomes (relatively) easy. The difficult part lies in realizing that some new entity, with a specific relationship to existing entities, will allow an entire body of observations to be explained with a much

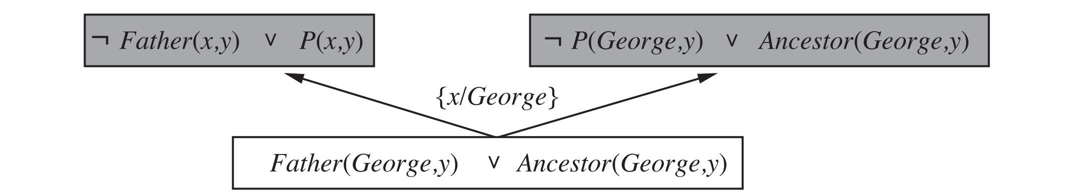

¬ Father(x,y) ∨ P(x,y)
¬ P(George,y) ∨ Ancestor(George,y)
{x/George}
Father(George,y) ∨ Ancestor(George,y)

Figure 19.14 An inverse resolution step that generates a new predicate P.

simpler and more elegant theory than previously existed.

As yet, ILP systems have not made discoveries on the level of Galileo or Joule, but their discoveries have been deemed publishable in the scientific literature. For example, in the *Journal of Molecular Biology*, Turcotte *et al.* (2001) describe the automated discovery of rules for protein folding by the ILP program PROGOL. Many of the rules discovered by PROGOL could have been derived from known principles, but most had not been previously published as part of a standard biological database. (See Figure 19.10 for an example.). In related work, Srinivasan *et al.* (1994) dealt with the problem of discovering molecular-structure-based rules for the mutagenicity of nitroaromatic compounds. These compounds are found in automobile exhaust fumes. For 80% of the compounds in a standard database, it is possible to identify four important features, and linear regression on these features outperforms ILP. For the remaining 20%, the features alone are not predictive, and ILP identifies relationships that allow it to outperform linear regression, neural nets, and decision trees. Most impressively, King *et al.* (2009) endowed a robot with the ability to perform molecular biology experiments and extended ILP techniques to include experiment design, thereby creating an autonomous scientist that actually discovered new knowledge about the functional genomics of yeast. For all these examples it appears that the ability both to represent relations and to use background knowledge contribute to ILP's high performance. The fact that the rules found by ILP can be interpreted by humans contributes to the acceptance of these techniques in biology journals rather than just computer science journals.

ILP has made contributions to other sciences besides biology. One of the most important is natural language processing, where ILP has been used to extract complex relational information from text. These results are summarized in Chapter 23.

19.6 SUMMARY

This chapter has investigated various ways in which prior knowledge can help an agent to learn from new experiences. Because much prior knowledge is expressed in terms of relational models rather than attribute-based models, we have also covered systems that allow learning of relational models. The important points are:

- The use of prior knowledge in learning leads to a picture of **cumulative learning**, in which learning agents improve their learning ability as they acquire more knowledge.
- Prior knowledge helps learning by eliminating otherwise consistent hypotheses and by

"filling in" the explanation of examples, thereby allowing for shorter hypotheses. These contributions often result in faster learning from fewer examples.

- Understanding the different logical roles played by prior knowledge, as expressed by **entailment constraints**, helps to define a variety of learning techniques.
- **Explanation-based learning** (EBL) extracts general rules from single examples by *explaining* the examples and generalizing the explanation. It provides a deductive method for turning first-principles knowledge into useful, efficient, special-purpose expertise.
- **Relevance-based learning** (RBL) uses prior knowledge in the form of determinations to identify the relevant attributes, thereby generating a reduced hypothesis space and speeding up learning. RBL also allows deductive generalizations from single examples.
- **Knowledge-based inductive learning** (KBIL) finds inductive hypotheses that explain sets of observations with the help of background knowledge.
- **Inductive logic programming** (ILP) techniques perform KBIL on knowledge that is expressed in first-order logic. ILP methods can learn relational knowledge that is not expressible in attribute-based systems.
- ILP can be done with a top-down approach of refining a very general rule or through a bottom-up approach of inverting the deductive process.
- ILP methods naturally generate new predicates with which concise new theories can be expressed and show promise as general-purpose scientific theory formation systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although the use of prior knowledge in learning would seem to be a natural topic for philosophers of science, little formal work was done until quite recently. *Fact, Fiction, and Forecast*, by the philosopher Nelson Goodman (1954), refuted the earlier supposition that induction was simply a matter of seeing enough examples of some universally quantified proposition and then adopting it as a hypothesis. Consider, for example, the hypothesis "All emeralds are grue," where *grue* means "green if observed before time t, but blue if observed thereafter." At any time up to t, we might have observed millions of instances confirming the rule that emeralds are grue, and no disconfirming instances, and yet we are unwilling to adopt the rule. This can be explained only by appeal to the role of relevant prior knowledge in the induction process. Goodman proposes a variety of different kinds of prior knowledge that might be useful, including a version of determinations called **overhypotheses**. Unfortunately, Goodman's ideas were never pursued in machine learning.

The **current-best-hypothesis** approach is an old idea in philosophy (Mill, 1843). Early work in cognitive psychology also suggested that it is a natural form of concept learning in humans (Bruner *et al.*, 1957). In AI, the approach is most closely associated with the work of Patrick Winston, whose Ph.D. thesis (Winston, 1970) addressed the problem of learning descriptions of complex objects. The **version space** method (Mitchell, 1977, 1982) takes a different approach, maintaining the set of *all* consistent hypotheses and eliminating those found to be inconsistent with new examples. The approach was used in the Meta-DENDRAL expert system for chemistry (Buchanan and Mitchell, 1978), and later in Mitchell's (1983) LEX system, which learns to solve calculus problems. A third influential thread was formed by the work of Michalski and colleagues on the AQ series of algorithms, which learned sets of logical rules (Michalski, 1969; Michalski *et al.*, 1986).

EBL had its roots in the techniques used by the STRIPS planner (Fikes *et al.*, 1972). When a plan was constructed, a generalized version of it was saved in a plan library and used in later planning as a **macro-operator**. Similar ideas appeared in Anderson's ACT* architecture, under the heading of **knowledge compilation** (Anderson, 1983), and in the SOAR architecture, as **chunking** (Laird *et al.*, 1986). **Schema acquisition** (DeJong, 1981), **analytical generalization** (Mitchell, 1982), and **constraint-based generalization** (Minton, 1984) were immediate precursors of the rapid growth of interest in EBL stimulated by the papers of Mitchell *et al.* (1986) and DeJong and Mooney (1986). Hirsh (1987) introduced the EBL algorithm described in the text, showing how it could be incorporated directly into a logic programming system. Van Harmelen and Bundy (1988) explain EBL as a variant of the **partial evaluation** method used in program analysis systems (Jones *et al.*, 1993).

Initial enthusiasm for EBL was tempered by Minton's finding (1988) that, without extensive extra work, EBL could easily slow down a program significantly. Formal probabilistic analysis of the expected payoff of EBL can be found in Greiner (1989) and Subramanian and Feldman (1990). An excellent survey of early work on EBL appears in Dietterich (1990).

ANALOGICAL REASONING

Instead of using examples as foci for generalization, one can use them directly to solve new problems, in a process known as **analogical reasoning**. This form of reasoning ranges from a form of plausible reasoning based on degree of similarity (Gentner, 1983), through a form of deductive inference based on determinations but requiring the participation of the example (Davies and Russell, 1987), to a form of "lazy" EBL that tailors the direction of generalization of the old example to fit the needs of the new problem. This latter form of analogical reasoning is found most commonly in **case-based reasoning** (Kolodner, 1993) and **derivational analogy** (Veloso and Carbonell, 1993).

Relevance information in the form of functional dependencies was first developed in the database community, where it is used to structure large sets of attributes into manageable subsets. Functional dependencies were used for analogical reasoning by Carbonell and Collins (1973) and rediscovered and given a full logical analysis by Davies and Russell (Davies, 1985; Davies and Russell, 1987). Their role as prior knowledge in inductive learning was explored by Russell and Grosof (1987). The equivalence of determinations to a restricted-vocabulary hypothesis space was proved in Russell (1988). Learning algorithms for determinations and the improved performance obtained by RBDTL were first shown in the FOCUS algorithm, due to Almuallim and Dietterich (1991). Tadepalli (1993) describes a very ingenious algorithm for learning with determinations that shows large improvements in learning speed.

The idea that inductive learning can be performed by inverse deduction can be traced to W. S. Jevons (1874), who wrote, "The study both of Formal Logic and of the Theory of Probabilities has led me to adopt the opinion that there is no such thing as a distinct method of induction as contrasted with deduction, but that induction is simply an inverse employment of deduction." Computational investigations began with the remarkable Ph.D. thesis by

Gordon Plotkin (1971) at Edinburgh. Although Plotkin developed many of the theorems and methods that are in current use in ILP, he was discouraged by some undecidability results for certain subproblems in induction. MIS (Shapiro, 1981) reintroduced the problem of learning logic programs, but was seen mainly as a contribution to the theory of automated debugging. Work on rule induction, such as the ID3 (Quinlan, 1986) and CN2 (Clark and Niblett, 1989) systems, led to FOIL (Quinlan, 1990), which for the first time allowed practical induction of relational rules. The field of relational learning was reinvigorated by Muggleton and Buntine (1988), whose CIGOL program incorporated a slightly incomplete version of inverse resolution and was capable of generating new predicates. The inverse resolution method also appears in (Russell, 1986), with a simple algorithm given in a footnote. The next major system was GOLEM (Muggleton and Feng, 1990), which uses a covering algorithm based on Plotkin's concept of relative least general generalization. ITOU (Rouveirol and Puget, 1989) and CLINT (De Raedt, 1992) were other systems of that era. More recently, PROGOL (Muggleton, 1995) has taken a hybrid (top-down and bottom-up) approach to inverse entailment and has been applied to a number of practical problems, particularly in biology and natural language processing. Muggleton (2000) describes an extension of PROGOL to handle uncertainty in the form of stochastic logic programs.

A formal analysis of ILP methods appears in Muggleton (1991), a large collection of papers in Muggleton (1992), and a collection of techniques and applications in the book by Lavra**u**c and D**u**zeroski (1994). Page and Srinivasan (2002) give a more recent overview of the field's history and challenges for the future. Early complexity results by Haussler (1989) suggested that learning first-order sentences was intractible. However, with better understanding of the importance of syntactic restrictions on clauses, positive results have been obtained even for clauses with recursion (D**u**zeroski *et al.*, 1992). Learnability results for ILP are surveyed by Kietz and D**u**zeroski (1994) and Cohen and Page (1995).

DISCOVERY SYSTEM

Although ILP now seems to be the dominant approach to constructive induction, it has not been the only approach taken. So-called **discovery systems** aim to model the process of scientific discovery of new concepts, usually by a direct search in the space of concept definitions. Doug Lenat's Automated Mathematician, or AM (Davis and Lenat, 1982), used discovery heuristics expressed as expert system rules to guide its search for concepts and conjectures in elementary number theory. Unlike most systems designed for mathematical reasoning, AM lacked a concept of proof and could only make conjectures. It rediscovered Goldbach's conjecture and the Unique Prime Factorization theorem. AM's architecture was generalized in the EURISKO system (Lenat, 1983) by adding a mechanism capable of rewriting the system's own discovery heuristics. EURISKO was applied in a number of areas other than mathematical discovery, although with less success than AM. The methodology of AM and EURISKO has been controversial (Ritchie and Hanna, 1984; Lenat and Brown, 1984).

Another class of discovery systems aims to operate with real scientific data to find new laws. The systems DALTON, GLAUBER, and STAHL (Langley *et al.*, 1987) are rule-based systems that look for quantitative relationships in experimental data from physical systems; in each case, the system has been able to recapitulate a well-known discovery from the history of science. Discovery systems based on probabilistic techniques—especially clustering algorithms that discover new categories—are discussed in Chapter 20.

EXERCISES

19.1 Show, by translating into conjunctive normal form and applying resolution, that the conclusion drawn on page 784 concerning Brazilians is sound.

19.2 For each of the following determinations, write down the logical representation and explain why the determination is true (if it is):

a. Design and denomination determine the mass of a coin.

b. For a given program, input determines output.

c. Climate, food intake, exercise, and metabolism determine weight gain and loss.

d. Baldness is determined by the baldness (or lack thereof) of one's maternal grandfather.

19.3 Would a probabilistic version of determinations be useful? Suggest a definition.

19.4 Fill in the missing values for the clauses C_1 or C_2 (or both) in the following sets of clauses, given that C is the resolvent of C_1 and C_2:

a. $C = \mathit{True} \Rightarrow P(A,B), C_1 = P(x,y) \Rightarrow Q(x,y), C_2 = ??.$

b. $C = \mathit{True} \Rightarrow P(A,B), C_1 = ??, C_2 = ??.$

c. $C = P(x,y) \Rightarrow P(x,f(y)), C_1 = ??, C_2 = ??.$

If there is more than one possible solution, provide one example of each different kind.

19.5 Suppose one writes a logic program that carries out a resolution inference step. That is, let $\mathit{Resolve}(c_1, c_2, c)$ succeed if c is the result of resolving c_1 and c_2. Normally, $\mathit{Resolve}$ would be used as part of a theorem prover by calling it with c_1 and c_2 instantiated to particular clauses, thereby generating the resolvent c. Now suppose instead that we call it with c instantiated and c_1 and c_2 uninstantiated. Will this succeed in generating the appropriate results of an inverse resolution step? Would you need any special modifications to the logic programming system for this to work?

19.6 Suppose that FOIL is considering adding a literal to a clause using a binary predicate P and that previous literals (including the head of the clause) contain five different variables.

a. How many functionally different literals can be generated? Two literals are functionally identical if they differ only in the names of the *new* variables that they contain.

b. Can you find a general formula for the number of different literals with a predicate of arity r when there are n variables previously used?

c. Why does FOIL not allow literals that contain no previously used variables?

19.7 Using the data from the family tree in Figure 19.11, or a subset thereof, apply the FOIL algorithm to learn a definition for the $\mathit{Ancestor}$ predicate.