10

MACHINE LEARNING: SYMBOL-BASED

The mind being, as I have declared, furnished with a great number of the simple ideas conveyed in by the senses, as they are found in exterior things, or by reflection on its own operations, takes notice, also, that a certain number of these simple ideas go constantly together . . . which, by inadvertency, we are apt afterward to talk of and consider as one simple idea.

—JOHN LOCKE, *Essay Concerning Human Understanding*

The mere observing of a thing is no use whatever. Observing turns into beholding, beholding into thinking, thinking into establishing connections, so that one may say that every attentive glance we cast on the world is an act of theorizing. However, this ought to be done consciously, with self criticism, with freedom, and to use a daring word, with irony.

—GOETHE

10.0 Introduction

The ability to learn must be part of any system that would claim to possess general intelligence. Indeed, in our world of symbols and interpretation, the very notion of an unchanging intellect seems a contradiction in terms. Intelligent agents must be able to change through the course of their interactions with the world, as well as through the experience of their own internal states and processes. We present four chapters on machine learning, reflecting four approaches to the problem: first, the symbol-based, second, the connectionist, third, the genetic/evoltionary, and finally, the dynamic/stochastic.

Learning is important for practical applications of artificial intelligence. Feigenbaum and McCorduck (1983) have called the “knowledge engineering bottleneck” the major obstacle to the widespread use of intelligent systems. This “bottleneck” is the cost and difficulty of building systems using the traditional knowledge acquisition techniques of Section 7.1. One solution to this problem would be for programs to begin with a minimal amount of knowledge and learn from examples, high-level advice, or their own explorations of the application domain.

Herbert Simon defines learning as:

any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population (Simon, 1983).

This definition, although brief, suggests many of the issues involved in developing programs that learn. Learning involves generalization from experience: performance should improve not only on the "repetition of the same task," but also on similar tasks in the domain. Because interesting domains tend to be large, a learner usually only examines a fraction of all possible examples; from this limited experience, the learner must generalize correctly to unseen instances of the domain. This is the problem of *induction*, and it is central to learning. In most learning problems, the available data are not sufficient to guarantee optimal generalization, no matter what algorithm is used. Learners must generalize heuristically, that is, they must select those aspects of their experience that are most likely to prove effective in the future. Such selection criteria are known as *inductive biases*.

Simon's definition describes learning as allowing the system to "perform better the second time." As the previous paragraph indicates, selecting the possible changes to a system that will allow it to improve is a difficult task. Learning research must address the possibility that changes may actually degrade performance. Preventing and detecting such problems is another issue for a learning algorithm.

Learning involves changes in the learner; this is clear. However, the exact nature of those changes and the best way to represent them are far from obvious. One approach models learning as the acquisition of explicitly represented domain knowledge. Based on its experience, the learner constructs or modifies expressions in a formal language, such as logic, and retains this knowledge for future use. *Symbolic approaches*, characterized by the algorithms of Chapter 10, are built on the assumption that the primary influence on the program's behavior is its collection of explicitly represented domain knowledge.

Neural or *connectionist networks*, Chapter 11, do not learn by acquiring sentences in a symbolic language. Like an animal brain, which consists of a large number of interconnected neurons, connectionist networks are systems of interconnected, artificial neurons. Knowledge is implicit in the organization and interaction of these neurons. Neural nets do not learn by adding representations to a knowledge base, but by modifying their overall structure to adapt to the contingencies of the world they inhabit. In Chapter 12, we consider *genetic* and *evolutionary learning*. This model of learning we have may be seen in the human and animal systems that have evolved towards equilibration with the world. This approach to learning through adaptation is reflected in genetic algorithms, genetic programming, and artificial life research.

Finally, in Chapter 13 we present *dynamic and probabilistic* approaches to learning. Many complex or not fully understood situations in the world, such as the generation and understanding of human language, can best be "understood" with probabilistic tools. Similarly, dynamical systems, like faults in a running mechanical engine, can be diagnosed.

In Chapter 10 we begin our exploration of machine learning with the symbol based approach where algorithms vary in their goals, in the available training data, and in the learning strategies and knowledge representation languages they employ. However, all of these algorithms learn by searching through a space of possible concepts to find an acceptable generalization. In Section 10.1, we outline a framework for symbol-based machine learning that emphasizes the common assumptions behind all of this work.

Although Section 10.1 outlines a variety of learning tasks, the full chapter focuses primarily on *inductive learning*. Induction, which is learning a generalization from a set of examples, is one of the most fundamental learning tasks. *Concept learning* is a typical inductive learning problem: given examples of some concept, such as “cat”, “soybean disease”, or “good stock investment”, we attempt to infer a definition that will allow the learner to correctly recognize future instances of that concept. Sections 10.2 and 10.3 examine two algorithms used for concept induction, *version space search* and *ID3*.

Section 10.4 considers the role of *inductive bias* in learning. The search spaces encountered in learning tend to be extremely large, even by the standards of search-based problem solving. These complexity problems are exacerbated by the problem of choosing among the different generalizations supported by the training data. Inductive bias refers to any method that a learning program uses to constrain the space of possible generalizations.

The algorithms of Sections 10.2 and 10.3 are data-driven. They use no prior knowledge of the learning domain but rely on large numbers of examples to define the essential properties of a general concept. Algorithms that generalize on the basis of patterns in training data are referred to as *similarity-based*.

In contrast to similarity-based methods, a learner may use prior knowledge of the domain to guide generalization. For example, humans do not require large numbers of examples to learn effectively. Often, a single example, analogy, or high-level bit of advice is sufficient to communicate a general concept. The effective use of such knowledge can help an agent to learn more efficiently, and with less likelihood of error. Section 10.5 examines *explanation-based learning*, learning by analogy and other techniques using prior knowledge to learn from limited training data.

The algorithms presented in Sections 10.2 through 10.5, though they differ in search strategies, representation languages, and the amount of prior knowledge used, all assume that the training data are classified by a teacher or some other means. The learner is told whether an instance is a positive or negative example of a target concept. This reliance on training instances of known classification defines the task of *supervised learning*.

Section 10.6 continues the study of induction by examining *unsupervised learning*, which addresses how an intelligent agent can acquire useful knowledge in the absence of correctly classified training data. *Category formation*, or *conceptual clustering*, is a fundamental problem in unsupervised learning. Given a set of objects exhibiting various properties, how can an agent divide the objects into useful categories? How do we even know whether a category will be useful? In this section, we examine CLUSTER/2 and COBWEB, two category formation algorithms.

Finally, in Section 10.7, we present reinforcement learning. Here, an agent is situated in an environment and receives feedback from that context. Learning requires the agent to act and then to interpret feedback from those actions. Reinforcement learning differs from supervised learning in that there is no “teacher” directly responding to each action; rather the agent itself must create a policy for interpreting all feedback. Reinforcement learning fits comfortably with a constructivist epistemology, as described in Section 16.2.

All learning presented in this chapter has one thing in common: it is seen as a variety of state space search. Even reinforcement learning derives a value function over a state space. We next outline a general search-based framework for work in machine learning.

10.1 A Framework for Symbol-Based Learning

Learning algorithms may be characterized along several dimensions, as is shown in Figure 10.1:

1. **The data and goals of the learning task**. One of the primary ways in which we characterize learning problems is according to the goals of the learner and the data it is given. The concept learning algorithms of Sections 10.2 and 10.3, for example, begin with a collection of positive (and usually negative) examples of a target class; the goal is to infer a general definition that will allow the learner to recognize future instances of the class. In contrast to the data-intensive approach taken by these algorithms, *explanation-based learning* (Section 10.5), attempts to infer a general concept from a single training example and a prior base of domain-specific knowledge. The conceptual clustering algorithms discussed in Section 10.6 illustrate another variation on the induction problem: instead of a set of training instances of known categorization, these algorithms begin with a set of unclassified instances. Their task is to discover categorizations that may have some utility to the learner.

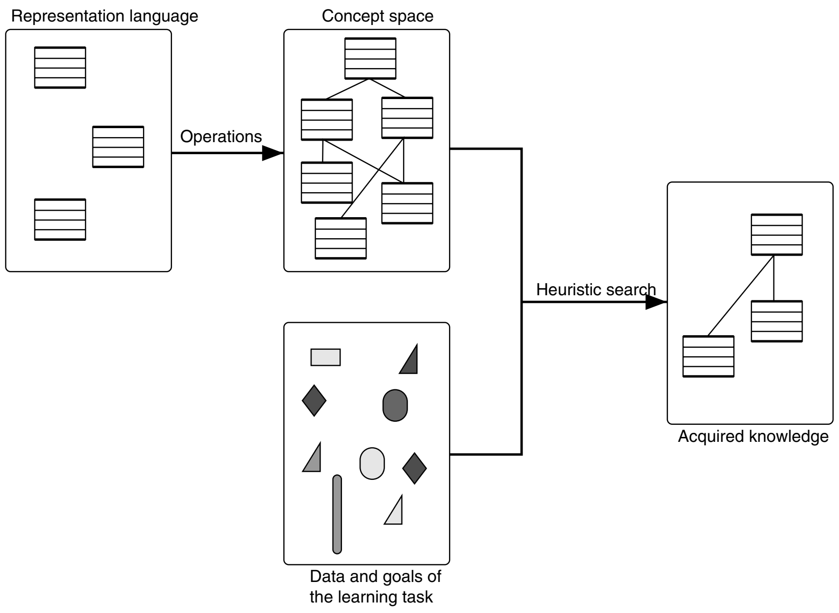


Figure 10.1 A general model of the learning process.

Examples are not the only source of training data. Humans, for instance, often learn from high-level advice. In teaching programming, professors generally tell their students that all loops must achieve a terminating condition. This advice, though correct, is not directly useful: it must be translated into specific rules for manipulating loop counters or logical conditions in a programming language. Analogies (Section 10.5.4) are another type of training data that must be correctly interpreted before they can be of use. If a teacher tells a student that electricity is like water, the student must infer the correct intent of the analogy: as water flows through a pipe, electricity flows through a wire. As with flowing water, we may measure the amount of electricity (amperage) and the pressure behind the flow (voltage). Unlike water, however, electricity does not make things wet or help us wash our hands. The interpretation of analogies involves finding the meaningful similarities and avoiding false or meaningless inferences.

We may also characterize a learning algorithm by the goal, or *target*, of the learner. The goal of many learning algorithms is a *concept*, or a general description of a class of objects. Learning algorithms may also acquire plans, problem-solving heuristics, or other forms of procedural knowledge.

The properties and quality of the training data itself are another dimension along which we classify learning tasks. The data may come from a teacher from the outside environment, or it may be generated by the program itself. Data may be reliable or may contain noise. It can be presented in a well-structured fashion or consist of unorganized data. It may include both positive and negative examples or only positive examples. Data may be readily available, the program may have to construct experiments, or perform some other form of data acquisition.

2. **The representation of learned knowledge.** Machine learning programs have made use of all the representation languages discussed in this text. For example, programs that learn to classify objects may represent these concepts as expressions in predicate calculus or they may use a structured representation such as frames or objects. Plans may be described as a sequence of operations or a triangle table. Heuristics may be represented as problem-solving rules.

   A simple formulation of the concept learning problem represents instances of a concept as conjunctive sentences containing variables. For example, two instances of “ball” (not sufficient to learn the concept) may be represented by:

   ```
   size(obj1, small) ∧ color(obj1, red) ∧ shape(obj1, round)
   size(obj2, large) ∧ color(obj2, red) ∧ shape(obj2, round)
   ```

   The general concept of “ball” could be defined by:

   ```
   size(X, Y) ∧ color(X, Z) ∧ shape(X, round)
   ```

   where any sentence that unifies with this general definition represents a ball.

3. **A set of operations.** Given a set of training instances, the learner must construct a generalization, heuristic rule, or plan that satisfies its goals. This requires the ability to manipulate representations. Typical operations include generalizing or

specializing symbolic expressions, adjusting the weights in a neural network, or otherwise modifying the program's representations.

In the concept learning example just introduced, a learner may generalize a definition by replacing constants with variables. If we begin with the concept:

size(obj1, small) $\wedge$ color(obj1, red) $\wedge$ shape(obj1, round)

replacing a single constant with a variable produces the generalizations:

size(obj1, X) $\wedge$ color(obj1, red) $\wedge$ shape(obj1, round)
size(obj1, small) $\wedge$ color(obj1, X) $\wedge$ shape(obj1, round)
size(obj1, small) $\wedge$ color(obj1, red) $\wedge$ shape(obj1, X)
size(X, small) $\wedge$ color(X, red) $\wedge$ shape(X, round)

4. **The concept space.** The representation language, together with the operations described above, defines a space of potential concept definitions. The learner must search this space to find the desired concept. The complexity of this concept space is a primary measure of the difficulty of a learning problem.
5. **Heuristic search.** Learning programs must commit to a direction and order of search, as well as to the use of available training data and heuristics to search efficiently. In our example of learning the concept "ball," a plausible algorithm may take the first example as a *candidate concept* and generalize it to include subsequent examples. For instance, on being given the single training example

   size(obj1, small) $\wedge$ color(obj1, red) $\wedge$ shape(obj1, round)

   the learner will make that example a candidate concept; this concept correctly classifies the only positive instance seen.

   If the algorithm is given a second positive instance

   size(obj2, large) $\wedge$ color(obj2, red) $\wedge$ shape(obj2, round)

   the learner may generalize the candidate concept by replacing constants with variables as needed to form a concept that matches both instances. The result is a more general candidate concept that is closer to our target concept of "ball."

   size(X, Y) $\wedge$ color(X, red) $\wedge$ shape(X, round)

Patrick Winston's work (1975*a*) on learning concepts from positive and negative examples illustrates these components. His program learns general definitions of structural concepts, such as "arch," in a blocks world. The training data is a series of positive and negative examples of the concept: examples of blocks world structures that fit in the category, along with *near misses*. The latter are instances that almost belong to the category but fail on one property or relation. The near misses enable the program to single out features that can be used to exclude negative instances from the target concept. Figure 10.2 shows positive examples and near misses for the concept "arch."

The program represents concepts as semantic networks, as in Figure 10.3. It learns by refining a candidate description of the target concept as training instances are presented. Winston's program refines candidate descriptions through generalization and specialization. Generalization changes the graph to let it accommodate new examples of the concept. Figure 10.3a shows an arch built of three bricks and a graph that describes it. The next training example, Figure 10.3b is an arch with a pyramid rather than a brick on top. This example does not match the candidate description. The program matches these graphs, attempting to find a partial isomorphism between them. The graph matcher uses the node names to guide the matching process. Once the program matches the graphs, it may detect differences between them. In Figure 10.3, the graphs match on all components except that the top element in the first graph is brick and the corresponding node of the second example is pyramid. Part of the program's background knowledge is a generalization hierarchy of these concepts, Figure 10.3c. The program generalizes the graph by replacing this node with the least common supertype of brick and pyramid; in this example, it is polygon. The result is the concept of Figure 10.3d.

When presented with a near miss, an example that differs from the target concept in a single property, the program specializes the candidate description to exclude the example. Figure 10.4a is a candidate description. It differs from the near miss of Figure 10.4b in the

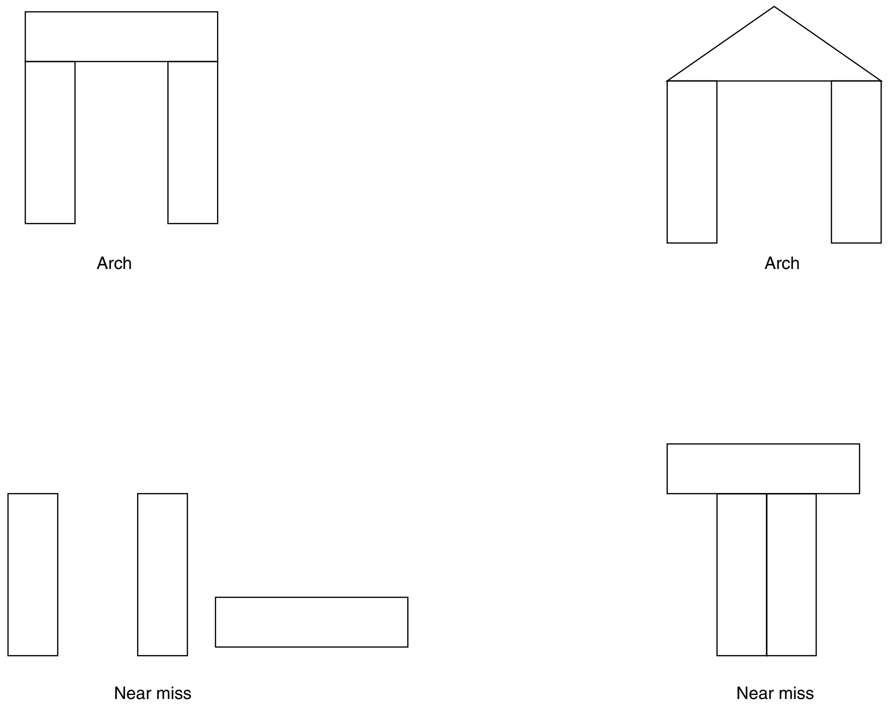
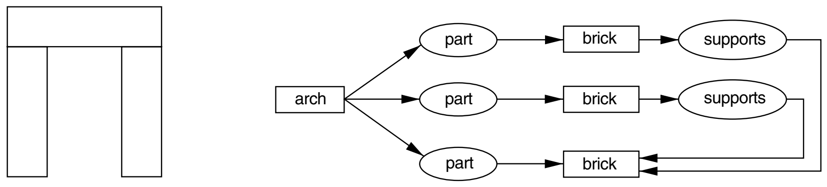


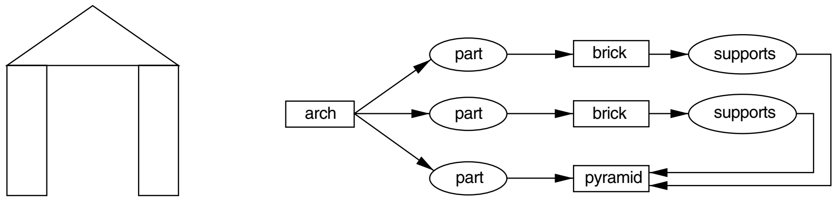
Figure 10.2 Examples and near misses for the concept arch.

touch relations of the near-miss example. The program specializes the graph by adding must-not-touch links to exclude the near miss, Figure 10.4c. Note that the algorithm depends heavily upon the closeness of the negative examples to the target concept. By differing from the goal in only a single property, a near miss helps the algorithm to determine exactly how to specialize the candidate concept.

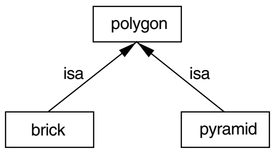
a. An example of an arch and its network description



b. An example of another arch and its network description



c. Given background knowledge that bricks and pyramids are both types of polygons



d. Generalization that includes both examples

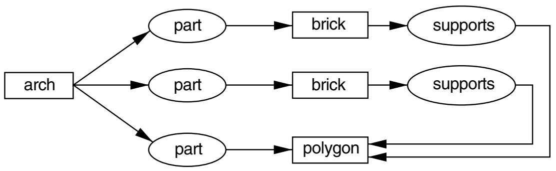


Figure 10.3 Generalization of descriptions to include multiple examples.

These operations—specializing a network by adding links and generalizing it by replacing node or link names with a more general concept—define a space of possible concept definitions. Winston's program performs a hill climbing search on the concept space guided by the training data. Because the program does not backtrack, its performance is highly sensitive to the order of the training examples; a bad ordering can lead the program to dead ends in the search space. Training instances must be presented to the program in an order that assists learning of the desired concept, much as a teacher organizes lessons to help a student learn. The quality and order of the training examples are also important to the program's graph matching algorithm; efficient matching requires that the graphs not be too dissimilar.

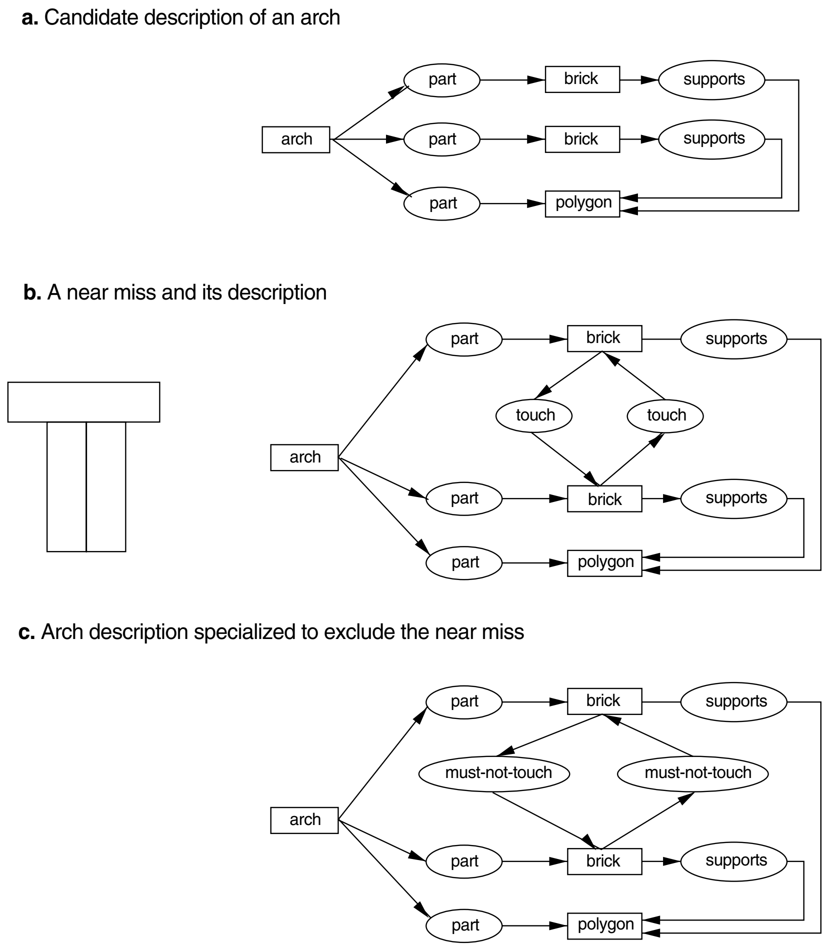


Figure 10.4 Specialization of a description to exclude a near miss. In 10.4c we add constraints to 10.4a so that it can t match with 10.4b.

Although an early example of inductive learning, Winston's program illustrates the features and problems shared by the majority of machine learning techniques of Chapter 10: the use of generalization and specialization operations to define a concept space, the use of data to guide search through that space, and the sensitivity of the learning algorithm to the quality of the training data. The next sections examine these problems and the techniques that machine learning has developed for their solution.

10.2 Version Space Search

Version space search (Mitchell 1978, 1979, 1982) illustrates the implementation of inductive learning as search through a concept space. Version space search takes advantage of the fact that generalization operations impose an ordering on the concepts in a space, and then uses this ordering to guide the search.

10.2.1 Generalization Operators and the Concept Space

Generalization and specialization are the most common types of operations for defining a concept space. The primary generalization operations used in machine learning are:

1. Replacing constants with variables. For example,

   color(ball, red)

   generalizes to

   color(X, red)

2. Dropping conditions from a conjunctive expression.

   shape(X, round) ∧ size(X, small) ∧ color(X, red)

   generalizes to

   shape(X, round) ∧ color(X, red)

3. Adding a disjunct to an expression.

   shape(X, round) ∧ size(X, small) ∧ color(X, red)

   generalizes to

   shape(X, round) ∧ size(X, small) ∧ (color(X, red) ∨ color(X, blue))

4. Replacing a property with its parent in a class hierarchy. If we know that primary_color is a superclass of red, then

color(X, red)

generalizes to

color(X, primary_color)

We may think of generalization in set theoretic terms: let P and Q be the sets of sentences matching the predicate calculus expressions p and q, respectively. Expression p is more general than q iff $P \supseteq Q$. In the above examples, the set of sentences that match color(X, red) contains the set of elements that match color(ball, red). Similarly, in example 2, we may think of the set of round, red things as a superset of the set of small, red, round things. Note that the "more general than" relationship defines a partial ordering on the space of logical sentences. We express this using the "≥" symbol, where $p \geq q$ means that p is more general than q. This ordering is a powerful source of constraints on the search performed by a learning algorithm.

We formalize this relationship through the notion of *covering*. If concept p is more general than concept q, we say that p *covers* q. We define the covers relation: let p(x) and q(x) be descriptions that classify objects as being positive examples of a concept. In other words, for an object x, p(x) → positive(x) and q(x) → positive(x). p covers q iff q(x) → positive(x) is a logical consequence of p(x) → positive(x).

For example, color(X, Y) covers color(ball, Z), which in turn covers color(ball, red). As a simple example, consider a domain of objects that have properties and values:

Sizes = {large, small}
Colors = {red, white, blue}
Shapes = {ball, brick, cube}

These objects can be represented using the predicate obj(Sizes, Color, Shapes). The generalization operation of replacing constants with variables defines the space of Figure 10.5. We may view inductive learning as searching this space for a concept that is consistent with all the training examples.

10.2.2 The Candidate Elimination Algorithm

This section presents three algorithms (Mitchell 1982) for searching the concept space. These algorithms rely upon the notion of a *version space*, which is the set of all concept descriptions consistent with the training examples. These algorithms work by reducing the size of the version space as more examples become available. The first two algorithms reduce the version space in a specific to general direction and a general to specific direction, respectively. The third algorithm, called *candidate elimination*, combines these approaches into a bi-directional search. We next describe and evaluate these algorithms.

These algorithms are data driven: they generalize based on regularities found in the training data. Also, in using training data of known classification, these algorithms perform a variety of *supervised learning*.

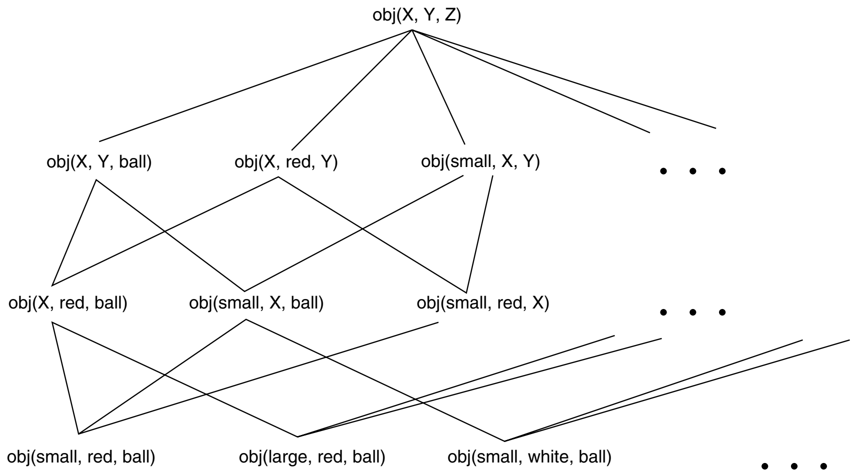


Figure 10.5 A concept space.

As with Winston's program for learning structural descriptions, version space search uses both positive and negative examples of the target concept. Although it is possible to generalize from positive examples only, negative examples are important in preventing the algorithm from overgeneralizing. Ideally, the learned concept must be general enough to cover all positive examples and also must be specific enough to exclude all negative examples. In the space of Figure 10.5, one concept that would cover all sets of exclusively positive instances would simply be obj(X, Y, Z). However, this concept is probably too general, because it implies that all instances belong to the target concept. One way to avoid overgeneralization is to generalize as little as possible to cover positive examples; another is to use negative instances to eliminate overly general concepts. As Figure 10.6 illustrates, negative instances prevent overgeneralization by forcing the learner to specialize concepts in order to exclude negative instances. The algorithms of this section use both of these techniques. We define *specific to general* search, for hypothesis set S, as:

```
Begin
Initialize S to the first positive training instance;
N is the set of all negative instances seen so far;

For each positive instance p
    Begin
    For every s ∈ S, if s does not match p, replace s with its most specific
        generalization that matchs p;
    Delete from S all hypotheses more general than some other hypothesis in S;
    Delete from S all hypotheses that match a previously observed negative
        instance in N;
    End;
```

```
For every negative instance n
    Begin
    Delete all members of S that match n;
    Add n to N to check future hypotheses for overgeneralization;
    End;
End
```

Specific to general search maintains a set, S, of *hypotheses*, or candidate concept definitions. To avoid overgeneralization, these candidate definitions are the *maximally specific generalizations* from the training data. A concept, c, is maximally specific if it covers all positive examples, none of the negative examples, and for any other concept, c′, that covers the positive examples, $c \leq c'$. Figure 10.7 shows an example of applying this algorithm to the version space of Figure 10.5. The specific to general version space search algorithm is built in Prolog in our auxiliary materials.

We may also search in a general to specific direction. This algorithm maintains a set, G, of *maximally general concepts* that cover all of the positive and none of the negative instances. A concept, c, is maximally general if it covers none of the negative training instances, and for any other concept, c′, that covers no negative training instance, $c \geq c'$. In this algorithm, negative instances lead to the specialization of candidate concepts; the algorithm uses positive instances to eliminate overly specialized concepts.

```
Begin
Initialize G to contain the most general concept in the space;
P contains all positive examples seen so far;

For each negative instance n
    Begin
    For each g ∈ G that matches n, replace g with its most general specializations
        that do not match n;
    Delete from G all hypotheses more specific than some other hypothesis in G;
    Delete from G all hypotheses that fail to match some positive example in P;
    End;

For each positive instance p
    Begin
    Delete from G all hypotheses that fail to match p;
    Add p to P;
    End;
End
```

Figure 10.8 shows an example of applying this algorithm to the version space of Figure 10.5. In this example, the algorithm uses background knowledge that size may have values {large, small}, color may have values {red, white, blue}, and shape may have values {ball, brick, cube}. This knowledge is essential if the algorithm is to specialize concepts by substituting constants for variables.

The *candidate elimination algorithm* combines these approaches into a bi-directional search. This bi-directional approach has a number of benefits for learning. The algorithm

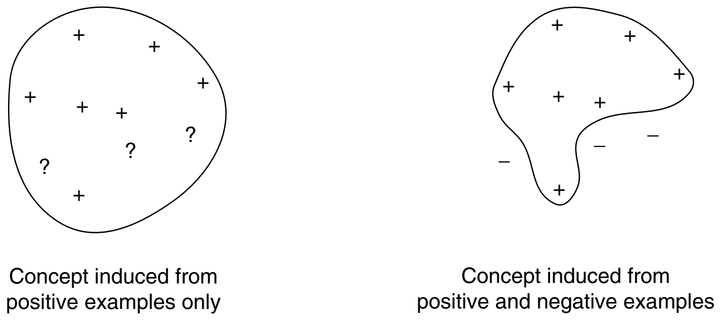

Concept induced from positive examples only
Concept induced from positive and negative examples

Figure 10.6 The role of negative examples in preventing overgeneralization.

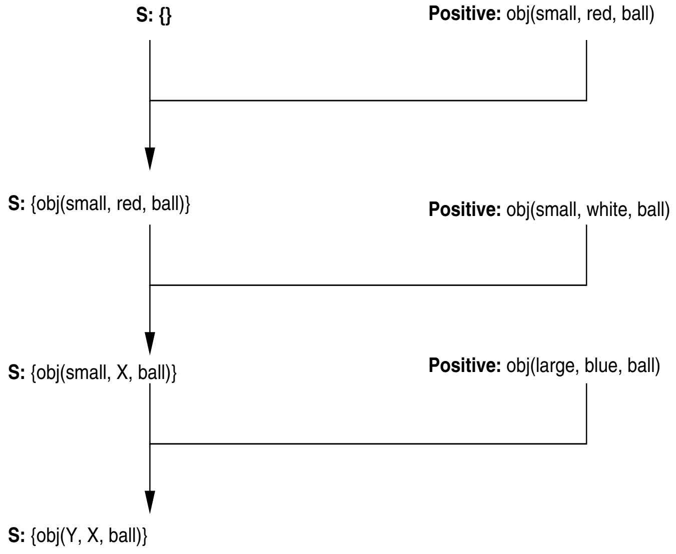

S: {}
Positive: obj(small, red, ball)
S: {obj(small, red, ball)}
Positive: obj(small, white, ball)
S: {obj(small, X, ball)}
Positive: obj(large, blue, ball)
S: {obj(Y, X, ball)}

Figure 10.7 Specific to general search of the version space learning the concept ball.

maintains two sets of candidate concepts: G, the set of maximally general candidate concepts, and S, the set of maximally specific candidates. The algorithm specializes G and generalizes S until they converge on the target concept. The algorithm is defined:

```
Begin
Initialize G to be the most general concept in the space;
Initialize S to the first positive training instance;
```

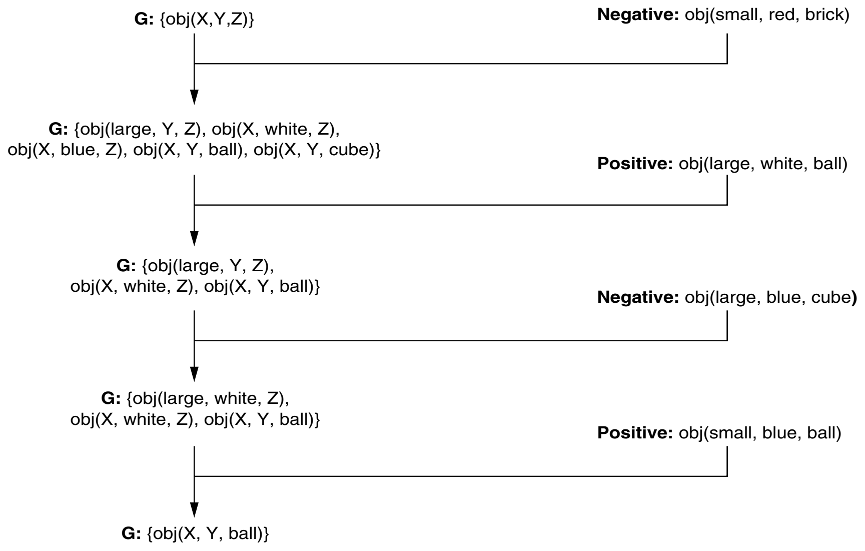
```
For each new positive instance p
      Begin
      Delete all members of G that fail to match p;
      For every s ∈ S, if s does not match p, replace s with its most specific
            generalizations that match p;
      Delete from S any hypothesis more general than some other hypothesis in S;
      Delete from S any hypothesis more general than some hypothesis in G;
      End;

For each new negative instance n
      Begin
      Delete all members of S that match n;
      For each g ∈ G that matches n, replace g with its most general specializations
            that do not match n;
      Delete from G any hypothesis more specific than some other hypothesis in G;
      Delete from G any hypothesis more specific than some hypothesis in S;
      End;

If G = S and both are singletons, then the algorithm has found a single concept that
      is consistent with all the data and the algorithm halts;

If G and S become empty, then there is no concept that covers all positive instances
      and none of the negative instances;

End
```



G: {obj(X,Y,Z)}
Negative: obj(small, red, brick)
G: {obj(large, Y, Z), obj(X, white, Z), obj(X, blue, Z), obj(X, Y, ball), obj(X, Y, cube)}
Positive: obj(large, white, ball)
G: {obj(large, Y, Z), obj(X, white, Z), obj(X, Y, ball)}
Negative: obj(large, blue, cube)
G: {obj(large, white, Z), obj(X, white, Z), obj(X, Y, ball)}
Positive: obj(small, blue, ball)
G: {obj(X, Y, ball)}

Figure 10.8 General to specific search of the version space learning the concept ball.

Figure 10.9 illustrates the behavior of the candidate elimination algorithm in searching the version space of Figure 10.5. Note that the figure does not show those concepts that were produced through generalization or specialization but eliminated as overly general or specific. We leave the elaboration of this part of the algorithm as an exercise and show a partial implementation in Prolog in our auxiliary materials.

Combining the two directions of search into a single algorithm has several benefits. The G and S sets summarize the information in the negative and positive training instances respectively, eliminating the need to save these instances. For example, after generalizing S to cover a positive instance, the algorithm uses G to eliminate concepts in S that do not cover any negative instances. Because G is the set of *maximally general* concepts that do not match any negative training instances, any member of S that is more general than any member of G must match some negative instance. Similarly, because S is the set of *maximally specific* generalizations that cover all positive instances, any new member of G that is more specific than a member of S must fail to cover some positive instance and may also be eliminated.

Figure 10.10 gives an abstract description of the candidate elimination algorithm. The "+" signs represent positive training instances; "–" signs indicate negative instances. The

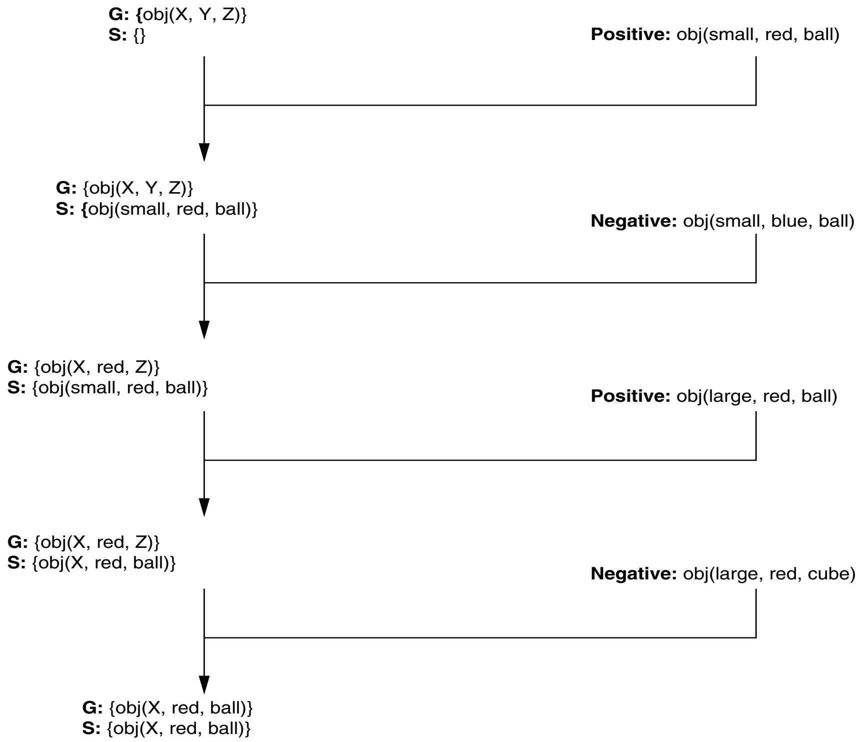

G: {obj(X, Y, Z)}
S: {}
Positive: obj(small, red, ball)
G: {obj(X, Y, Z)}
S: {obj(small, red, ball)}
Negative: obj(small, blue, ball)
G: {obj(X, red, Z)}
S: {obj(small, red, ball)}
Positive: obj(large, red, ball)
G: {obj(X, red, Z)}
S: {obj(X, red, ball)}
Negative: obj(large, red, cube)
G: {obj(X, red, ball)}
S: {obj(X, red, ball)}

Figure 10.9 The candidate elimination algorithm learning the concept red ball.

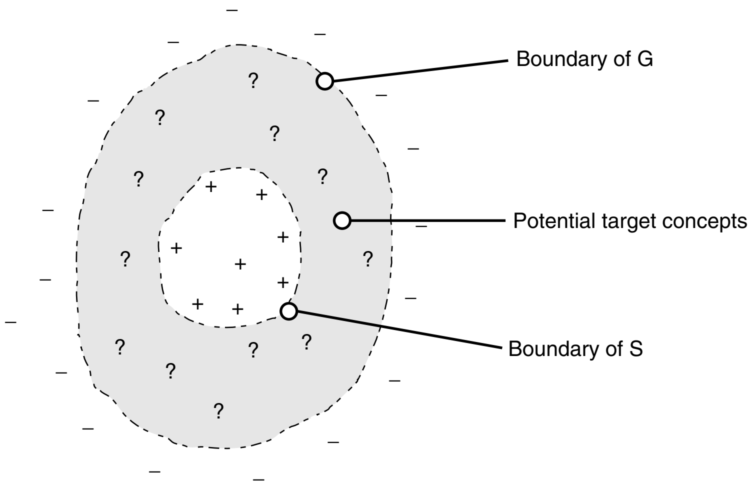


Figure 10.10 Converging boundaries of the G and S sets in the candidate elimination algorithm.

innermost circle encloses the set of known positive instances covered by the concepts in S. The outermost circle encloses the instances covered by G; any instance outside this circle is negative. The shaded portion of the graphic contains the target concept, along with concepts that may be overly general or specific (the ?s). The search "shrinks" the outermost concept as necessary to exclude negative instances; it "expands" the innermost concept to include new positive instances. Eventually, the two sets converge on the target concept. In this fashion, candidate elimination can detect when it has found a single, consistent target concept. When both G and S converge to the same concept the algorithm may halt. If G and S become empty, then there is no concept that will cover all positive instances and none of the negative instances. This may occur if the training data is inconsistent or if the goal concept may not be expressed in the representation language (Section 10.2.4).

An interesting aspect of candidate elimination is its incremental nature. An incremental learning algorithm accepts training instances one at a time, forming a usable, although possibly incomplete, generalization after each example. This contrasts with batch algorithms, (see for example ID3, Section 10.3), which require all training examples to be present before they may begin learning. Even before the candidate elimination algorithm converges on a single concept, the G and S sets provide usable constraints on that concept: if c is the goal concept, then for all $g \in G$ and $s \in S$, $s \leq c \leq g$. Any concept that is more general than some concept in G will cover negative instances; any concept that is more specific than some concept in S will fail to cover some positive instances. This suggests that instances that have a "good fit" with the concepts bounded by G and S are at least plausible instances of the concept.

In the next section, we clarify this intuition with an example of a program that uses candidate elimination to learn search heuristics. LEX (Mitchell et al. 1983) learns heuristics for solving symbolic integration problems. Not only does this work demonstrate the use of G and S to define partial concepts, but it also illustrates such additional issues as the complexities of learning multistep tasks, credit/blame assignment, and the relationship between the learning and problem-solving components of a complex system.

10.2.3 LEX: Inducing Search Heuristics

LEX learns heuristics for solving symbolic integration problems. LEX integrates algebraic expressions through heuristic search, beginning with the expression to be integrated and searching for its goal: an expression that contains no integral signs. The learning component of the system uses data from the problem solver to induce heuristics that improve the problem solver's performance.

LEX searches a space defined by operations on algebraic expressions. Its operators are the typical transformations used in performing integration. They include:

OP1: $\int r\, f(x)\, dx \rightarrow r \int f(x)\, dx$
OP2: $\int u\, dv \rightarrow uv - \int v\, du$
OP3: $1 * f(x) \rightarrow f(x)$
OP4: $\int (f_1(x) + f_2(x))\, dx \rightarrow \int f_1(x)\, dx + \int f_2(x)\, dx$

Operators are rules, whose left-hand side defines when they may be applied. Although the left-hand side defines the circumstances under which the operator may be used, it does not include heuristics for when the operator *should* be used. LEX must learn usable heuristics through its own experience. Heuristics are expressions of the form:

If the current problem state matches P then apply operator O with bindings B.

For example, a typical heuristic that LEX might learn is:

If a problem state matches $\int$ x transcendental(x) dx,
then apply OP2 with bindings
u = x
dv = transcendental(x) dx

Here, the heuristic suggests applying integration by parts to solve the integral of x times some transcendental function, e.g., a trigonometric function in x.

LEX's language for representing concepts consists of the symbols described in Figure 10.11. Note that the symbols exist in a generalization hierarchy, with any symbol matching any of its descendants in the hierarchy. LEX generalizes expressions by replacing a symbol with its ancestor in this hierarchy.

For example, given the expression:

$\int 3x \cos(x)\, dx$

LEX may replace cos with trig. This yields the expression:

∫ 3x trig(x) dx

Alternatively, it may replace 3 with the symbol k, which represents any integer:

∫ kx cos(x) dx

Figure 10.12 shows a version space for OP2 as defined by these generalizations.

The overall architecture of LEX consists of four components:

1. a *generalizer* that uses candidate elimination to find heuristics

2. a *problem solver* that produces traces of problem solutions

3. a *critic* that produces positive and negative instances from a problem trace

4. a *problem generator* that produces new candidate problems

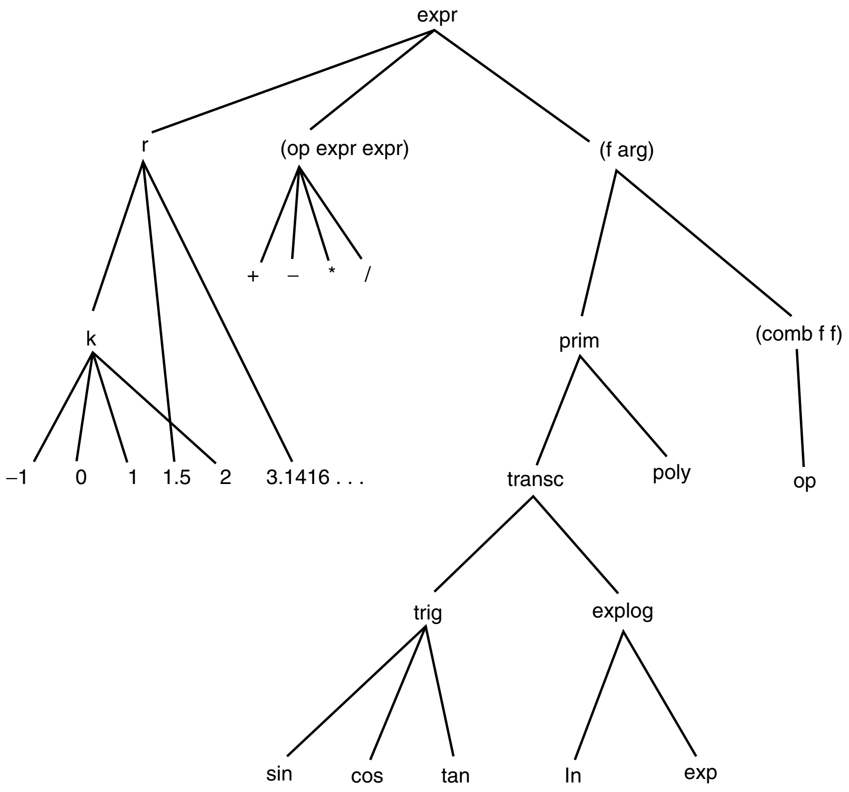


Figure 10.11 A portion of LEX s hierarchy of symbols.

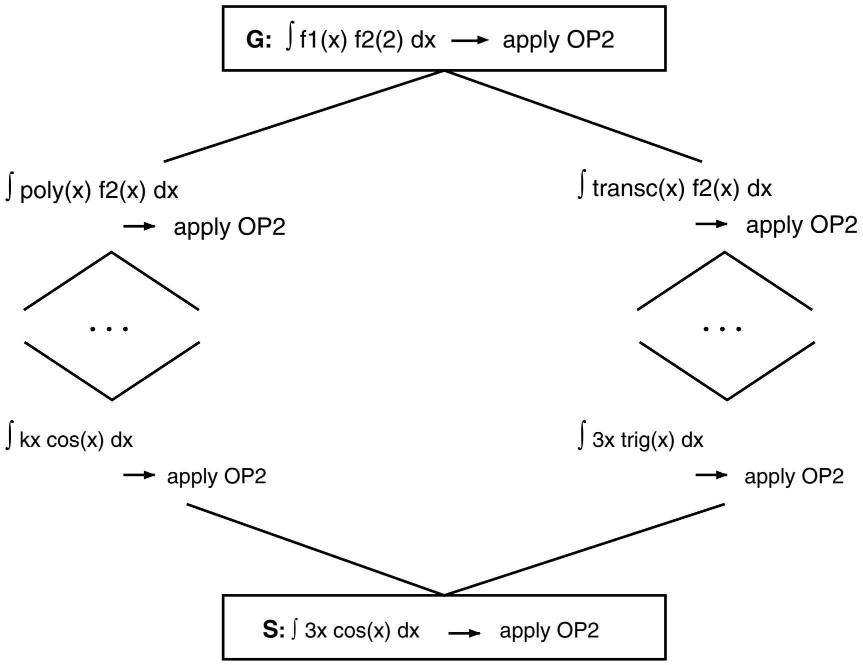

G: ∫ f1(x) f2(2) dx → apply OP2
∫ poly(x) f2(x) dx → apply OP2
∫ transc(x) f2(x) dx → apply OP2
. . .
. . .
∫ kx cos(x) dx → apply OP2
∫ 3x trig(x) dx → apply OP2
S: ∫ 3x cos(x) dx → apply OP2

Figure 10.12 A version space for OP2, adapted from Mitchell et al. (1983).

LEX maintains a set of version spaces. Each version space is associated with an operator and represents a partially learned heuristic for that operator. The generalizer updates these version spaces using positive and negative examples of the operator's application, as generated by the critic. On receiving a positive instance, LEX determines whether a version space associated with that operator includes the instance. A version space includes a positive instance if the instance is covered by some of the concepts in G. LEX then uses the positive instance to update that heuristic. If no existing heuristic matches the instance, LEX creates a new version space, using that instance as the first positive example. This can lead to creating multiple version spaces, for different heuristics, and one operator.

LEX's problem solver builds a tree of the space searched in solving an integration problem. It limits the CPU time the problem solver may use to solve a problem. LEX performs best-first search, using its own developing heuristics. An interesting aspect of LEX's performance is its use of G and S as partial definitions of a heuristic. If more than one operator may apply to a given state, LEX chooses the one that exhibits the highest degree of partial match to the problem state. Degree of partial match is defined as the percentage of all the concepts included between G and S that match the current state. Because the computational expense of testing the state against all such candidate concepts would be prohibitive, LEX estimates the degree of match as the percentage of entries actually in G and S that match the state. Note that performance should improve steadily as LEX improves its heuristics. Empirical results have confirmed this conjecture.

LEX obtains positive and negative examples of operator applications from the solution trace generated by the problem solver. In the absence of a teacher, LEX must classify operator applications as positive or negative; this is an example of the *credit assignment* problem. When learning is undertaken in the context of multistep problem solving, it is often unclear which action in a sequence should be given responsibility for the result. If a problem solver arrives at a wrong answer, how do we know which of several steps actually caused the error? LEX's critic approaches this problem by assuming that the solution trace returned by the problem solver represents a shortest path to a goal. LEX classifies applications of operators along this (assumed) shortest path as positive instances; operator applications that diverge from this path are treated as negative instances.

However, in treating the problem solver's trace as a shortest path solution, the critic must address the fact that LEX's evolving heuristics are not guaranteed to be admissible (Section 4.3). The solution path found by the problem solver may not actually be a shortest path solution. To ensure that it has not erroneously classified an operator application as negative, LEX first extends the paths begun by such operators to make sure they do not lead to a better solution. Usually, a problem solution produces 2 to 20 training instances. LEX passes these positive and negative instances on to the generalizer, which uses them to update the version spaces for the associated operators.

The problem generator is the least developed part of the program. Although various strategies were used to automate problem selection, most examples involved hand-chosen instances. However, a problem generator was constructed that explored a variety of strategies. One approach generates instances that were covered by the partial heuristics for two different operators, in order to make LEX learn to discriminate between them.

Empirical tests show that LEX is effective in learning useful heuristics. In one test, LEX was given 5 test problems and 12 training problems. Before training, it solved the 5 test problems in an average of 200 steps; these solutions used no heuristics to guide the search. After developing heuristics from the 12 training problems, it solved these same test problems in an average of 20 steps.

LEX addresses a number of issues in learning, including such problems as credit assignment, the selection of training instances, and the relationship between the problem solving and generalization components of a learning algorithm. LEX also underscores the importance of an appropriate representation for concepts. Much of LEX's effectiveness stems from the hierarchical organization of concepts. This hierarchy is small enough to constrain the space of potential heuristics and to allow efficient search, while being rich enough to represent effective heuristics.

10.2.4 Evaluating Candidate Elimination

The candidate elimination algorithm demonstrates the way in which knowledge representation and state space search can be applied to the problem of machine learning. However, as with most important research, the algorithm should not be evaluated in terms of its successes alone. It raises problems that continue to form a sizeable portion of machine learning's research agenda.

Search-based learning, like all search problems, must deal with the combinatorics of problem spaces. When the candidate elimination algorithm performs breadth-first search, it can be inefficient. If an application is such that G and S grow excessively, it may be useful to develop heuristics for pruning states from G and S, implementing best first, or even better a beam search (see Section 4.5) of the space.

Another approach to this problem, discussed in Section 10.4, involves using an *inductive bias* to further reduce the size of the concept space. Such biases constrain the language used to represent concepts. LEX imposed a bias through the choice of concepts in its generalization hierarchy. Though not complete, LEX's concept language was strong enough to capture many effective heuristics; of equal importance, it reduced the size of the concept space to manageable proportions. Biased languages are essential in reducing the complexity of the concept space, but they may leave the learner incapable of representing the concept it is trying to learn. In this case, candidate elimination would fail to converge on the target concept, leaving G and S empty. This trade-off between expressiveness and efficiency is an essential issue in learning.

Failure of the algorithm to converge may also be due to some noise or inconsistency in the training data. The problem of learning from noisy data is particularly important in realistic applications, where data cannot be guaranteed to be complete or consistent. Candidate elimination is not at all noise resistant. Even a single misclassified training instance can prevent the algorithm from converging on a consistent concept. One solution to this problem maintains multiple G and S sets. In addition to the version space derived from all training instances, it maintains additional spaces based on all but one of the training instances, all but two of the training instances, etc. If G and S fail to converge, the algorithm can examine these alternatives to find those that remain consistent. Unfortunately, this approach leads to a proliferation of candidate sets and is too inefficient to be practical in most cases.

Another issue raised by this research is the role of prior knowledge in learning. LEX's concept hierarchy summarized a great deal of knowledge about algebra; this knowledge was essential to the algorithm's performance. Can greater amounts of domain knowledge make learning even more effective? Section 10.5 examines this problem by using prior domain knowledge to guide generalizations.

An important contribution of this work is its explication of the relationship between knowledge representation, generalization, and search in inductive learning. Although candidate elimination is only one of many learning algorithms, it raises general questions concerning complexity, expressiveness, and the use of knowledge and data to guide generalization. These problems are central to all machine learning algorithms; we continue to address them throughout this chapter.

10.3 The ID3 Decision Tree Induction Algorithm

ID3 (Quinlan 1986*a*), like candidate elimination, induces concepts from examples. It is particularly interesting for its representation of learned knowledge, its approach to the management of complexity, its heuristic for selecting candidate concepts, and its potential for handling noisy data. ID3 represents concepts as *decision trees*, a representation that allows us to determine the classification of an object by testing its values for certain properties.

For example, consider the problem of estimating an individual's credit risk on the basis of such properties as credit history, current debt, collateral, and income. Table 10.1 lists a sample of individuals with known credit risks. The decision tree of Figure 10.13 represents the classifications in Table 10.1, in that this tree can correctly classify all the objects in the table. In a decision tree, each internal node represents a test on some property, such as credit history or debt; each possible value of that property corresponds to a branch of the tree. Leaf nodes represent classifications, such as low or moderate risk. An individual of unknown type may be classified by traversing this tree: at each internal node, test the individual's value for that property and take the appropriate branch. This continues until reaching a leaf node and the object's classification.

Note that in classifying any given instance, this tree does not use all the properties present in Table 10.1. For instance, if a person has a good credit history and low debt, we may, according to the tree, ignore her collateral and income and classify her as a low risk. In spite of omitting certain tests, this tree correctly classifies all the examples.

NO.	RISK	CREDIT HISTORY	DEBT	COLLATERAL	INCOME
1.	high	bad	high	none	$0 to $15k
2.	high	unknown	high	none	$15 to $35k
3.	moderate	unknown	low	none	$15 to $35k
4.	high	unknown	low	none	$0 to $15k
5.	low	unknown	low	none	over $35k
6.	low	unknown	low	adequate	over $35k
7.	high	bad	low	none	$0 to $15k
8.	moderate	bad	low	adequate	over $35k
9.	low	good	low	none	over $35k
10.	low	good	high	adequate	over $35k
11.	high	good	high	none	$0 to $15k
12.	moderate	good	high	none	$15 to $35k
13.	low	good	high	none	over $35k
14.	high	bad	high	none	$15 to $35k

Table 10.1 Data from credit history of loan applications

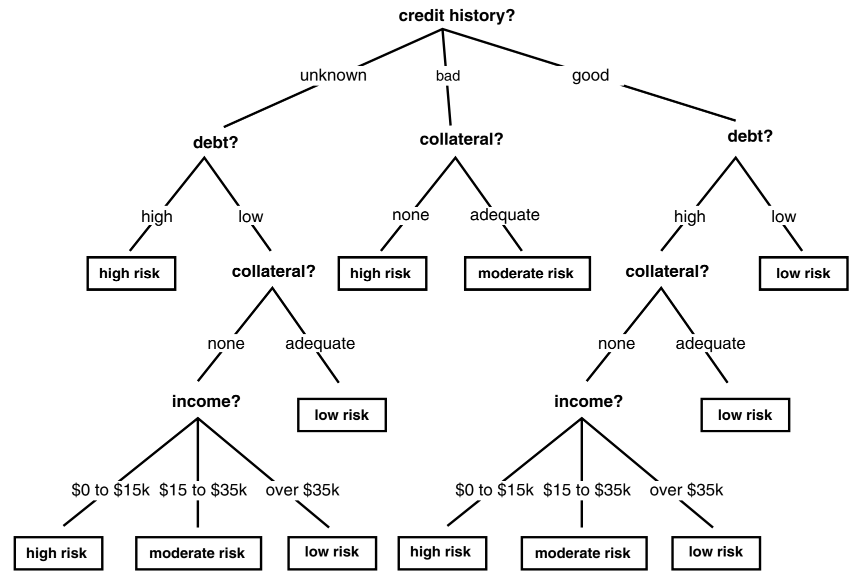


Figure 10.13 A decision tree for credit risk assessment.

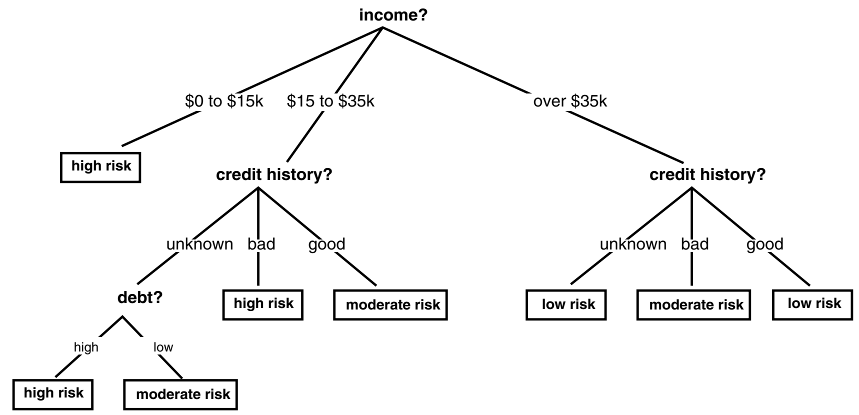


Figure 10.14 A simplified decision tree for credit risk assessment.

In general, the size of the tree necessary to classify a given set of examples varies according to the order with which properties are tested. Figure 10.14 shows a tree that is considerably simpler than that of Figure 10.13 but that also classifies correctly the examples in Table 10.1.

Given a set of training instances and a number of different decision trees that correctly classify them, we may ask which tree has the greatest likelihood of correctly classifying unseen instances of the population. The ID3 algorithm assumes that this is the simplest decision tree that covers all the training examples. The rationale for this assumption is the time-honored heuristic of preferring simplicity and avoiding unnecessary assumptions. This principle, known as *Occam's Razor*, was first articulated by the medieval logician William of Occam in 1324:

> It is vain to do with more what can be done with less. . . . Entities should not be multiplied beyond necessity.

A more contemporary version of Occam's Razor argues that we should always accept the simplest answer that correctly fits our data. In this case, it is the smallest decision tree that correctly classifies all given examples.

Although Occam's Razor has proven itself as a general heuristic for all manner of intellectual activity, its use here has a more specific justification. If we assume that the given examples are sufficient to construct a valid generalization, then our problem becomes one of distinguishing the necessary properties from the extraneous ones. The simplest decision tree that covers all the examples should be the least likely to include unnecessary constraints. Although this idea is intuitively appealing, it is an assumption that must be empirically tested; Section 10.3.3 presents some of these empirical results. Before examining these results, however, we present the ID3 algorithm for inducing decision trees from examples.

10.3.1 Top-Down Decision Tree Induction

ID3 constructs decision trees in a top-down fashion. Note that for any property, we may partition the set of training examples into disjoint subsets, where all the examples in a partition have a common value for that property. ID3 selects a property to test at the current node of the tree and uses this test to partition the set of examples; the algorithm then recursively constructs a subtree for each partition. This continues until all members of the partition are in the same class; that class becomes a leaf node of the tree. Because the order of tests is critical to constructing a simple decision tree, ID3 relies heavily on its criteria for selecting the test at the root of each subtree. To simplify our discussion, this section describes the algorithm for constructing decision trees, assuming an appropriate test selection function. In Section 10.3.2, we present the selection heuristic of the ID3 algorithm.

For example, consider the way in which ID3 constructs the tree of Figure 10.14 from Table 10.1. Beginning with the full table of examples, ID3 selects income as the root property using the selection function described in Section 10.3.2. This partitions the example set as shown in Figure 10.15, with the elements of each partition being listed by their number in the table.

The induction algorithm begins with a sample of the correctly classified members of the target categories. ID3 then constructs a decision tree according to the following algorithm:

```
function induce_tree (example_set, Properties)

begin
if all entries in example_set are in the same class
   then return a leaf node labeled with that class
   else if Properties is empty
      then return leaf node labeled with disjunction of all classes in example_set
      else begin
         select a property, P, and make it the root of the current tree;
         delete P from Properties;
            for each value, V, of P,
               begin
                  create a branch of the tree labeled with V;
                  let partition$_v$ be elements of example_set with values V for property P;
                  call induce_tree(partition$_v$, Properties), attach result to branch V
               end
      end
end
```

ID3 applies the induce_tree function recursively to each partition. For example, the partition {1, 4, 7, 11} consists entirely of high-risk individuals; ID3 creates a leaf node accordingly. ID3 selects the credit history property as the root of the subtree for the partition {2, 3, 12, 14}. In Figure 10.16, credit history further divides this four element partition into {2, 3}, {14}, and {12}. Continuing to select tests and construct subtrees in this fashion, ID3 eventually produces the tree of Figure 10.14. The reader can work through the rest of this construction; we present a Lisp implementation in the auxiliary material.

Before presenting ID3's test selection heuristic, it is worth examining the relationship between the tree construction algorithm and our view of learning as search through a concept space. We may think of the set of all possible decision trees as defining a version space. Our operations for moving through this space consist of adding tests to a tree. ID3 implements a form of greedy search in the space of all possible trees: it adds a subtree to the current tree and continues its search; it does not backtrack. This makes the algorithm highly efficient; it also makes it dependent upon the criteria for selecting properties to test.

10.3.2 Information Theoretic Test Selection

We may think of each property of an instance as contributing a certain amount of information to its classification. For example, if our goal is to determine the species of an animal, the discovery that it lays eggs contributes a certain amount of information to that goal. ID3 measures the information gained by making each property the root of the current subtree. It then picks the property that provides the greatest information gain.

Information theory (Shannon 1948) provides a mathematical basis for measuring the information content of a message. We may think of a message as an instance in a universe of possible messages; the act of transmitting a message is the same as selecting one of these possible messages. From this point of view, it is reasonable to define the information

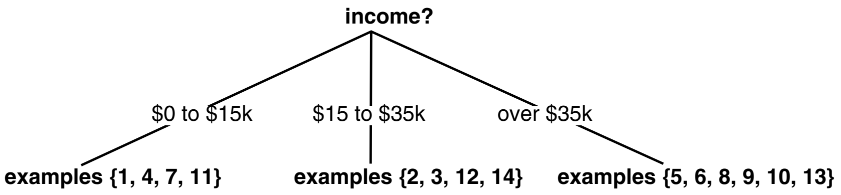


Figure 10.15 A partially constructed decision tree.

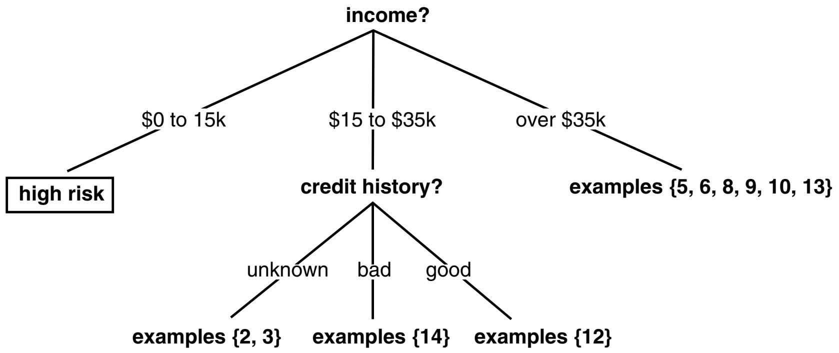


Figure 10.16 Another partially constructed decision tree.

content of a message as depending upon both the size of this universe and the frequency with which each possible message occurs.

The importance of the number of possible messages is evident in an example from gambling: compare a message correctly predicting the outcome of a spin of the roulette wheel with one predicting the outcome of a toss of an honest coin. Because roulette can have more outcomes than a coin toss, a message concerning its outcome is of more value to us: winning at roulette also pays better than winning at a coin toss. Consequently, we should regard this message as conveying more information.

The influence of the probability of each message on the amount of information is evident in another gambling example. Assume that I have rigged a coin so that it will come up heads $^3/_4$ of the time. Because I already know enough about the coin to wager correctly $^3/_4$ of the time, a message telling me the outcome of a given toss is worth less to me than it would be for an honest coin.

Shannon formalized this by defining the amount of information in a message as a function of the probability of occurrence p of each possible message, namely, $-\log_2 p$. Given a universe of messages, $M = \{m_1, m_2, \ldots m_n\}$ and a probability, $p(m_i)$, for the occurrence of each message, the expected information content of a message M is given by:

$$I[M] = \left(\sum_{i=1}^{n} -p(m_i)\ \log_2\ (p(m_i)) \right) = E[-\log_2 p(m_i)]$$

The information in a message is measured in bits. For example, the information content of a message telling the outcome of the flip of an honest coin is:

$$\begin{aligned} I[\text{Coin toss}] &= -p(\text{heads})\log_2(p(\text{heads})) - p(\text{tails})\log_2(p(\text{tails})) \\ &= -\frac{1}{2}\log_2\left(\frac{1}{2}\right) - \frac{1}{2}\log_2\left(\frac{1}{2}\right) \\ &= 1\ \text{bit} \end{aligned}$$

However, if the coin has been rigged to come up heads 75 per cent of the time, then the information content of a message is:

$$\begin{aligned} I[\text{Coin toss}] &= -\frac{3}{4}\log_2\left(\frac{3}{4}\right) - \frac{1}{4}\log_2\left(\frac{1}{4}\right) \\ &= -\frac{3}{4}*(-0.415) - \frac{1}{4}*(-2) \\ &= 0.811\ \text{bits} \end{aligned}$$

This definition formalizes many of our intuitions about the information content of messages. Information theory is widely used in computer science and telecommunications, including such applications as determining the information-carrying capacity of communications channels, developing data compression algorithms, and developing noise resistant communication strategies. ID3 uses information theory to select the test that gives the greatest information gain in classifying the training examples.

We may think of a decision tree as conveying information about the classification of examples in the decision table; the information content of the tree is computed from the probabilities of the different classifications. For example, if we assume that all the examples in Table 10.1 occur with equal probability, then:

$$p(\text{risk is high}) = {}^6/_{14},\ p(\text{risk is moderate}) = {}^3/_{14},\ p(\text{risk is low}) = {}^5/_{14}$$

It follows that the distribution described in Table 10.1, $D_{9.1}$, and, consequently, any tree that covers those examples, is:

$$\begin{aligned} I[D_{9.1}] &= -\frac{6}{14}\log_2\left(\frac{6}{14}\right) - \frac{3}{14}\log_2\left(\frac{3}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) \\ &= -\frac{6}{14}*(-1.222) - \frac{3}{14}*(-2.222) - \frac{5}{14}*(-1.485) \\ &= 1.531\ \text{bits} \end{aligned}$$

The information gain provided by making a test at the root of the current tree is equal to the total information in the tree minus the amount of information needed to complete the classification after performing the test. The amount of information needed to complete the tree is defined as the weighted average of the information in all its subtrees. We compute the weighted average by multiplying the information content of each subtree by the percentage of the examples present in that subtree and summing these products.

Assume a set of training instances, C. If we make property P, with n values, the root of the current tree, this will partition C into subsets, $\{C_1, C_2, \ldots C_n\}$. The expected information needed to complete the tree after making P the root is:

$$E[P] = \sum_{i=1}^{n} \frac{|C_i|}{|C|} I[C_i]$$

The gain from property P is computed by subtracting the expected information to complete the tree from the total information content of the tree:

$$\text{gain}(P) = I[C] - E[P]$$

In the example of Table 10.1, if we make income the property tested at the root of the tree, this partitions the table of examples into the partitions $C_1 = \{1,4,7,11\}$, $C_2 = \{2,3,12,14\}$, and $C_3 = \{5,6,8,9,10,13\}$. The expected information needed to complete the tree is:

$$\begin{aligned} E[\text{income}] &= \frac{4}{14} * I[C_1] + \frac{4}{14} * I[C_2] + \frac{6}{14} * I[C_3] \\ &= \frac{4}{14} * 0.0 + \frac{4}{14} * 1.0 + \frac{6}{14} * 0.650 \\ &= 0.564 \text{ bits} \end{aligned}$$

The information gain for the distribution of Table 10.1 is:

$$\begin{aligned} \text{gain(income)} &= I[D_{9.1}] - E[\text{income}] \\ &= 1.531 - 0.564 \\ &= 0.967 \text{ bits} \end{aligned}$$

Similarly, we may show that

$$\begin{aligned} \text{gain(credit history)} &= 0.266 \\ \text{gain(debt)} &= 0.063 \\ \text{gain (collateral)} &= 0.206 \end{aligned}$$

Because income provides the greatest information gain, ID3 will select it as the root of the tree. The algorithm continues to apply this analysis recursively to each subtree until it has completed the tree.

10.3.3 Evaluating ID3

Although the ID3 algorithm produces simple decision trees, it is not obvious that such trees will be effective in predicting the classification of unknown examples. ID3 has been evaluated in both controlled tests and applications and has proven to work well in practice.

Quinlan, for example, has evaluated ID3's performance on the problem of learning to classify boards in a chess endgame (Quinlan 1983). The endgame involved white, playing with a king and a rook, against black, playing with a king and a knight. ID3's goal was to learn to recognize boards that led to a loss for black within three moves. The attributes were different high-level properties of boards, such as "an inability to move the king safely." The test used 23 such attributes.

Once board symmetries were taken into account, the entire problem domain consisted of 1.4 million different boards, of which 474,000 were a loss for black in three moves. ID3 was tested by giving it a randomly selected training set and then testing it on 10,000 different boards, also randomly selected. Quinlan's tests gave the results found in Table 10.2. The predicted maximum errors were derived from a statistical model of ID3's behavior in the domain. For further analysis and details see Quinlan (1983).

These results are supported by further empirical studies and by anecdotal results from further applications. Variations of ID3 have been developed to deal with such problems as noise and excessively large training sets. For more details, see Quinlan (1986*a*, *b*).

Size of Training Set	Percentage of Whole Universe	Errors in 10,000 Trials	Predicted Maximum Errors
200	0.01	199	728
1,000	0.07	33	146
5,000	0.36	8	29
25,000	1.79	6	7
125,000	8.93	2	1

Table 10.2 The evaluation of ID3

10.3.4 Decision Tree Data Issues: Bagging, Boosting

Quinlan (1983) was the first to suggest the use of information theory to produce subtrees in decision tree learning and his work was the basis for our presentation. Our examples were clean, however, and their use straightforward. There are a number of issues that we did not address, each of which often occurs in a large data set:

1. The data is bad. This can happen when two (or more) identical attribute sets give different results. What can we do if we have no a priori reason to get rid of data?

2. Data from some attribute sets is missing, perhaps because it is too expensive to obtain. Do we extrapolate? Can we create a new value “unknown?” How can we smooth over this irregularity?

3. Some of the attribute sets are continuous. We handled this by breaking the continuous value “income” into convenient subsets of values, and then used these groupings. Are there better approaches?

4. The data set may be too large for the learning algorithm. How do you handle this?

Addressing these issues produced new generations of decision tree learning algorithms after ID3. The most notable of these is C4.5 (Quinlan 1996). These issues also led to techniques such as bagging and boosting. Since the data for classifier learning systems are attribute-value vectors or instances, it is tempting to manipulate the data to see if different classifiers are produced.

Bagging produces replicate training sets by sampling with replacement from the training instances. *Boosting* uses all instances at each replication, but maintains a weight for each instance in the training set. This weight is intended to reflect that vector’s importance. When the weights are adjusted, different classifiers are produced, since the weights cause the learner to focus on different instances. In either case, the multiple classifiers produced are combined by voting to form a composite classifier. In bagging, each component classifier has the same vote, while boosting assigns different voting strengths to the component classifiers on the basis of their accuracy.

In working with very large sets of data, it is common to divide the data into subsets, build the decision tree on one subset, and then test its accuracy on other subsets. The literature on decision tree learning is now quite extensive, with a number of data sets on-line, and a number of empirical results published showing the results of using various versions of decision tree algorithms on this data.

Finally, it is straightforward to convert a decision tree into a comparable rule set. What we do is make each possible path through the decision tree into a single rule. The pattern for the rule, its left-hand side, consists of the decisions leading to the leaf node. The action or right-hand side is the leaf node or outcome of the tree. This rule set may be further customized to capture subtrees within the decision tree. This rule set may then be run to classify new data.

10.4 Inductive Bias and Learnability

So far, our discussion has emphasized the use of empirical data to guide generalization. However, successful induction also depends upon prior knowledge and assumptions about the nature of the concepts being learned. *Inductive bias* refers to any criteria a learner uses to constrain the concept space or to select concepts within that space. In the next section, we examine the need for bias and the types of biases that learning programs typically employ. Section 10.4.2 introduces theoretical results in quantifying the effectiveness of inductive biases.

10.4.1 Inductive Bias

Learning spaces tend to be large; without some way of pruning them, search-based learning would be a practical impossibility. For example, consider the problem of learning a classification of bit strings (strings of 0s and 1s) from positive and negative examples. Because a classification is simply a subset of the set of all possible strings, the total number of possible classifications is equal to the power set, or set of all subsets, of the entire population. If there are m instances, there are 2^m possible classifications. But for strings of n bits, there are 2^n different strings. Thus, there are 2 to the power 2^n different classifications of bit strings of length n. For $n = 50$, this number is larger than the number of molecules in the known universe! Without some heuristic constraints, it would be impossible for a learner to effectively search such spaces in all but the most trivial domains.

Another reason for the necessity of bias is the nature of inductive generalization itself. Generalization is not truth preserving. For example, if we encounter an honest politician, are we justified in assuming that all politicians are honest? How many honest politicians must we encounter before we are justified in making this assumption? Hume discussed this problem, known as the problem of induction, several hundred years ago:

> You say that the one proposition is an inference from the other; but you must confess that the inference is not intuitive, neither is it demonstrative. Of what nature is it then? To say it is experimental is begging the question. For all inferences from experience suppose, as their foundation, that the future will resemble the past and that similar powers will be conjoined with similar sensible qualities (Hume 1748).

Incidentally, in the eighteenth century Hume's work was seen as an intellectual threat, especially to the religious community's attempts to mathematically prove the existence and attributes of the deity. Among the counter theories proposed to rescue "certainty" was that of Rev. Bayes, an English cleric, see Chapter 5, 9, and 13. But back to induction.

In inductive learning, the training data are only a subset of all instances in the domain; consequently, any training set may support many different generalizations. In our example of a bit string classifier, assume that the learner has been given the strings {1100, 1010} as positive examples of some class of strings. Many generalizations are consistent with these examples: the set of all strings beginning with "1" and ending with "0," the set of all strings beginning with "1," the set of all strings of even parity, or any other subset of the entire population that includes {1100, 1010}. What can the learner use to choose from these generalizations? The data alone are not sufficient; all of these choices are consistent with the data. The learner must make additional assumptions about "likely" concepts.

In learning, these assumptions often take the form of heuristics for choosing a branch of the search space. The information theoretic test selection function used by ID3 (Section 10.3.2) is an example of such a heuristic. ID3 performs a hill-climbing search through the space of possible decision trees. At each stage of the search, it examines all the tests that could be used to extend the tree and chooses the test that gains the most information. This is a “greedy” heuristic: it favors branches of the search space that seem to move the greatest distance toward a goal state.

This heuristic allows ID3 to search efficiently the space of decision trees, and it also addresses the problem of choosing plausible generalizations from limited data. ID3 assumes that the smallest tree that correctly classifies all the given examples will be the most likely to classify future training instances correctly. The rationale for this assumption is that small trees are less likely to make assumptions not supported by the data. If the training set is large enough and truly representative of the population, such trees should include all and only the essential tests for determining class membership. As discussed in Section 10.3.3, empirical evaluations have shown this assumption to be quite justified. This preference for simple concept definitions is used in a number of learning algorithms, such as the CLUSTER/2 algorithm of Section 10.6.2.

Another form of inductive bias consists of syntactic constraints on the representation of learned concepts. Such biases are not heuristics for selecting a branch of the concept space. Instead, they limit the size of the space itself by requiring that learned concepts be expressed in a constrained representation language. Decision trees, for example, are a much more constrained language than full predicate calculus. The corresponding reduction in the size of the concept space is essential to ID3’s efficiency.

An example of a syntactic bias that might prove effective in classifying bit strings would limit concept descriptions to patterns of symbols from the set {0, 1, #}. A pattern defines the class of all matching strings, where matching is determined according to the following rules:

> If the pattern has a 0 in a certain position, then the target string must have a 0 in the corresponding position.
>
> If the pattern has a 1 in a certain position, then the target string must have a 1 in the corresponding position.
>
> A # in a given position can match either a 1 or a 0.

For example, the pattern, “1##0” defines the set of strings {1110, 1100, 1010, 1000}.

Considering only those classes that could be represented as a single such pattern reduces the size of the concept space considerably. For strings of length n, we may define 3^n different patterns. This is considerably smaller than the 2 to the power 2^n possible concepts in the unconstrained space. This bias also allows straightforward implementation of version space search, where generalization involves replacing a 1 or a 0 in a candidate pattern with a #. However, the cost we incur for this bias is the inability to represent (and consequently learn) certain concepts. For example, a single pattern of this type cannot represent the class of all strings of even parity.

This trade-off between expressiveness and efficiency is typical. LEX, for example, does not distinguish between odd or even integers in its taxonomy of symbols. Consequently, it cannot learn any heuristic that depends upon this distinction. Although work has been done in programs that can change their bias in response to data (Utgoff 1986), most learning programs assume a fixed inductive bias.

Machine learning has explored a number of representational biases:

Conjunctive biases restrict learned knowledge to conjunctions of literals. This is particularly common because the use of disjunction in concept descriptions creates problems for generalization. For example, assume that we allow arbitrary use of disjuncts in the representation of concepts in the candidate elimination algorithm. Because the maximally specific generalization of a set of positive instances is simply the disjunction of all the instances, the learner will not generalize at all. It will add disjuncts *ad infinitum*, implementing a form of rote learning (Mitchell 1980).

Limitations on the number of disjuncts. Purely conjunctive biases are too limited for many applications. One approach that increases the expressiveness of a representation while addressing the problems of disjunction is to allow a small, bounded number of disjuncts.

Feature vectors are a representation that describes objects as a set of features whose values differ from object to object. The objects presented in Table 10.1 are represented as sets of features.

Decision trees are a concept representation that has proven effective in the ID3 algorithm.

Horn clauses require a restriction on the form of implications that has been used in automated reasoning as well as by a number of programs for learning rules from examples. We present Horn clauses in detail in Section 14.2.

In addition to the syntactic biases discussed in this section, a number of programs use domain-specific knowledge to consider the known or assumed semantics of the domain. Such knowledge can provide an extremely effective bias. Section 10.5 examines these knowledge-based approaches. However, before considering the role of knowledge in learning, we briefly examine theoretical results quantifying inductive bias. We also present a summary discussion of inductive bias in learning systems in Section 16.2.

10.4.2 The Theory of Learnability

The goal of inductive bias is to restrict the set of target concepts in such a way that we may both search the set efficiently and find high-quality concept definitions. An interesting body of theoretical work addresses the problem of quantifying the effectiveness of an inductive bias.

We define the quality of concepts in terms of their ability to correctly classify objects that were not included in the set of training instances. It is not hard to write a learning algorithm that produces concepts that will correctly classify all the examples that it has seen; rote learning would suffice for this. However, due to the large number of instances in most domains, or the fact that some instances are not available, algorithms can only afford to examine a portion of the possible examples. Thus, the performance of a learned concept on new instances is critically important. In testing learning algorithms, we generally divide the set of all instances into nonintersecting sets of training instances and test instances. After training a program on the training set, we test it on the test set.

It is useful to think of efficiency and correctness as properties of the language for expressing concepts, i.e., the inductive bias, rather than of a particular learning algorithm. Learning algorithms search a space of concepts; if this space is manageable and contains concepts that perform well, then any reasonable learning algorithm should find these definitions; if the space is highly complex, an algorithm's success will be limited. An extreme example will clarify this point.

The concept of ball is learnable, given a suitable language for describing the properties of objects. After seeing a relatively small number of balls, a person will be able to define them concisely: balls are round. Contrast this with a concept that is not learnable: suppose a team of people runs around the planet and selects a set of several million objects entirely at random, calling the resulting class bunch_of_stuff. Not only would a concept induced from any sample of bunch_of_stuff require an extremely complex representation, but it also is unlikely that this concept would correctly classify unseen members of the set.

These observations make no assumption about the learning algorithms used, just that they can find a concept in the space consistent with the data. ball is learnable because we can define it in terms of a few features: the concept can be expressed in a biased language. Attempting to describe the concept bunch_of_stuff would require a concept definition as long as the list of all the properties of all the objects in the class.

Thus, rather than defining learnability in terms of specific algorithms, we define it in terms of the language used to represent concepts. Also, to achieve generality, we do not define learnability over specific problem domains, such as learning bunch_of_stuff. Instead we define it in terms of the syntactic properties of the concept definition language.

In defining learnability, we must not only take efficiency into account; we must also deal with the fact that we have limited data. In general we cannot hope to find the exactly correct concept from a random sample of instances. Rather, just as in estimating the mean of a set in statistics, we try to find a concept which is very likely to be nearly correct. Consequently, the correctness of a concept is the probability, over the entire population of instances, that it will correctly classify an instance.

In addition to the correctness of learned concepts, we must also consider the likelihood that an algorithm will find such concepts. That is, there is a small chance that the samples we see are so atypical that learning is impossible. Thus, a particular distribution of positive instances, or a particular training set selected from these instances, may or may not be sufficient to select a high-quality concept. We are therefore concerned with two probabilities: the probability that our samples are not atypical and the probability that the algorithm will find a quality concept, the normal estimation error. These two probabilities are bounded by δ and ε, respectively, in the definition of PAC learability we give below.

To summarize, learnability is a property of concept spaces and is determined by the language required to represent concepts. In evaluating these spaces, we must take into account both the probability that the data is by coincidence quite impoverished and the probability with which the resulting concept will correctly classify the unseen instances.

Valiant (1984) has formalized these intuitions in the theory of probably approximately correct (PAC) learning.

A class of concepts is *PAC learnable* if an algorithm exists that executes efficiently and has a high probability of finding an *approximately correct* concept. By approximately correct, we mean that the concept correctly classifies a high percentage of new instances. Thus, we require both that the algorithm find, with a high probability, a concept that is nearly correct and that the algorithm itself be efficient. An interesting aspect of this definition is that it does not necessarily depend upon the distribution of positive examples in the instance space. It depends upon the nature of the concept language, that is, the bias, and the desired degree of correctness. Finally, by making assumptions about the example distributions, it is often possible to get better performance, that is, to make do with fewer samples than the theory requires.

Formally, Valiant defines PAC learnability as follows. Let C be a set of concepts c and X a set of instances. The concepts may be algorithms, patterns, or some other means of dividing X into positive and negative instances. C is PAC learnable if there exists an algorithm with the following properties:

1. If for concept error ε and failure probability δ, there exists an algorithm which, given a random sample of instances of size $n = |X|$ polynomial in $1/\varepsilon$, and $1/\delta$, the algorithm produces a concept c, an element of C, such that the probability that c has a generalization error greater than ε is less than δ. That is, for y drawn from the same distribution that the samples in X were drawn from:

   $P[P[y \text{ is misclassified by } c] \geq \varepsilon] \leq \delta.$

2. The execution time for the algorithm is polynomial in n, $1/\varepsilon$, and $1/\delta$.

Using this definition of PAC learnability, researchers have shown the tractability of several inductive biases. For example, Valiant (1984) proves that the class of k-CNF expressions is learnable. k-CNF expressions are sentences in conjunctive normal form with a bound on the number of disjuncts; expressions are formed of the conjunction of clauses, $c_1 \wedge c_2 \wedge \ldots c_n$, where each c_i is the disjunction of no more than k literals. This theoretical result supports the common restriction of concepts to conjunctive form used in many learning algorithms. We do not duplicate the proof here but refer the reader to Valiant's paper, where he proves this result, along with the learnability of other biases. For additional results in learnability and inductive bias see Haussler (1988) and Martin (1997).

10.5 Knowledge and Learning

ID3 and the candidate elimination algorithm generalize on the basis of regularities in training data. Such algorithms are often referred to as *similarity based*, in that generalization is primarily a function of similarities across training examples. The biases employed by these algorithms are limited to syntactic constraints on the form of learned knowledge; they make no strong assumptions about the semantics of the domains. In this section, we examine algorithms, such as *explanation-based learning*, that use prior domain knowledge to guide generalization.

Initially, the idea that prior knowledge is necessary for learning seems contradictory. However, both machine learning and cognitive scientist researchers have made a case for exactly that notion, arguing that the most effective learning occurs when the learner already has considerable knowledge of the domain. One argument for the importance of knowledge in learning is the reliance of similarity-based learning techniques on relatively large amounts of training data. Humans, in contrast, can form reliable generalizations from as few as a single training instance, and many practical applications require that a learning program do the same.

Another argument for the importance of prior knowledge recognizes that any set of training examples can support an unlimited number of generalizations, most of which are either irrelevant or nonsensical. Inductive bias is one means of making this distinction. In this section, we examine algorithms that go beyond purely syntactic biases to consider the role of strong domain knowledge in learning.

10.5.1 Meta-DENDRAL

Meta-DENDRAL (Buchanan and Mitchell 1978) is one of the earliest and still one of the best examples of the use of knowledge in inductive learning. Meta-DENDRAL acquires rules to be used by the DENDRAL program for analyzing mass spectrographic data. DENDRAL infers the structure of organic molecules from their chemical formula and mass spectrographic data.

A mass spectrograph bombards molecules with electrons, causing some of the chemical bonds to break. Chemists measure the weight of the resulting pieces and interpret these results to gain insight into the structure of the compound. DENDRAL employs knowledge in the form of rules for interpreting mass spectrographic data. The premise of a DENDRAL rule is a graph of some portion of a molecular structure. The conclusion of the rule is that graph with the location of the cleavage indicated.

Meta-DENDRAL infers these rules from spectrographic results on molecules of known structure. Meta-DENDRAL is given the structure of a known compound, along with the mass and relative abundance of the fragments produced by spectrography. It interprets these, constructing an account of where the breaks occurred. These explanations of breaks in specific molecules are used as examples for constructing general rules.

In determining the site of a cleavage in a training run, DENDRAL uses a “half-order theory” of organic chemistry. This theory, though not powerful enough to support the direct construction of DENDRAL rules, does support the interpretation of cleavages within known molecules. The half-order theory consists of rules, constraints, and heuristics such as:

Double and triple bonds do not break.

Only fragments larger than two carbon atoms show up in the data.

Using the half-order theory, meta-DENDRAL constructs explanations of the cleavage. These explanations indicate the likely sites of cleavages along with possible migrations of atoms across the break.

These explanations become the set of positive instances for a rule induction program. This component induces the constraints in the premises of DENDRAL rules through a general to specific search. It begins with a totally general description of a cleavage: $X_1 * X_2$. This pattern means that a cleavage, indicated by the asterisk, can occur between any two atoms. It specializes the pattern by:

adding atoms: $X_1 * X_2 \rightarrow X_3 - X_1 * X_2$

where the "–" operator indicates a chemical bond, or

instantiating atoms or attributes of atoms: $X_1 * X_2 \rightarrow C * X_2$

Meta-DENDRAL learns from positive examples only and performs a hill-climbing search of the concept space. It prevents overgeneralization by limiting candidate rules to cover only about half of the training instances. Subsequent components of the program evaluate and refine these rules, looking for redundant rules or modifying rules that may be overly general or specific.

The strength of meta-DENDRAL is in its use of domain knowledge to change raw data into a more usable form. This gives the program noise resistance, through the use of its theory to eliminate extraneous or potentially erroneous data, and the ability to learn from relatively few training instances. The insight that training data must be so interpreted to be fully useful is the basis of explanation-based learning.

10.5.2 Explanation-Based Learning

Explanation-based learning uses an explicitly represented domain theory to construct an explanation of a training example, usually a proof that the example logically follows from the theory. By generalizing from the explanation of the instance, rather than from the instance itself, explanation-based learning filters noise, selects relevant aspects of experience, and organizes training data into a systematic and coherent structure.

There are several alternative formulations of this idea. For example, the STRIPS program for representing general operators for planning (see Section 8.4) has exerted a powerful influence on this research (Fikes et al. 1972). Meta-DENDRAL, as we saw in Section 10.5.1, established the power of theory-based interpretation of training instances. More recently, a number of authors (DeJong and Mooney 1986, Minton 1988) have proposed alternative formulations of this idea. The *Explanation-Based Generalization* algorithm of Mitchell et al. (1986) is also typical of the genre. In this section, we examine a variation of the explanation-based learning (EBL) algorithm developed by DeJong and Mooney (1986).

EBL begins with:

1. *A target concept*. The learner's task is to determine an effective definition of this concept. Depending upon the specific application, the target concept may be a classification, a theorem to be proven, a plan for achieving a goal, or a heuristic for a problem solver.
2. *A training example*, an instance of the target.
3. *A domain theory*, a set of rules and facts that are used to explain how the training example is an instance of the goal concept.
4. *Operationality criteria*, some means of describing the form that concept definitions may take.

To illustrate EBL, we present an example of learning about when an object is a cup. This is a variation of a problem explored by Winston et al. (1983) and adapted to explanation-based learning by Mitchell et al. (1986). The target concept is a rule that may be used to infer whether an object is a cup:

```
premise(X) → cup(X)
```

where premise is a conjunctive expression containing the variable X.

Assume a domain theory that includes the following rules about cups:

```
liftable(X) ∧ holds_liquid(X) → cup(X)
part(Z, W) ∧ concave(W) ∧ points_up(W) → holds_liquid(Z)
light(Y) ∧ part(Y, handle) → liftable(Y)
small(A) → light(A)
made_of(A, feathers) → light(A)
```

The training example is an instance of the goal concept. That is, we are given:

```
cup(obj1)
small(obj1)
part(obj1, handle)
owns(bob, obj1)
part(obj1, bottom)
part(obj1, bowl)
points_up(bowl)
concave(bowl)
color(obj1, red)
```

Finally, assume the operationality criteria require that target concepts be defined in terms of observable, structural properties of objects, such as part and points_up. We may provide domain rules that enable the learner to infer whether a description is operational, or we may simply list operational predicates.

Using this theory, a theorem prover may construct an explanation of why the example is indeed an instance of the training concept: a proof that the target concept logically follows from the example, as in the first tree in Figure 10.17. Note that this explanation eliminates such irrelevant concepts as color(obj1, red) from the training data and captures those aspects of the example known to be relevant to the goal.

The next stage of explanation-based learning generalizes the explanation to produce a concept definition that may be used to recognize other cups. EBL accomplishes this by substituting variables for those constants in the proof tree that depend solely on the particular training instance, as in Figure 10.17. Based on the generalized tree, EBL defines a new rule whose conclusion is the root of the tree and whose premise is the conjunction of the leaves:

$$\text{small}(X) \wedge \text{part}(X, \text{handle}) \wedge \text{part}(X, W) \wedge \text{concave}(W) \wedge \text{points\_up}(W) \rightarrow \text{cup}(X).$$

In constructing a generalized proof tree, our goal is to substitute variables for those constants that are part of the training instance while retaining those constants and constraints that are part of the domain theory. In this example, the constant handle

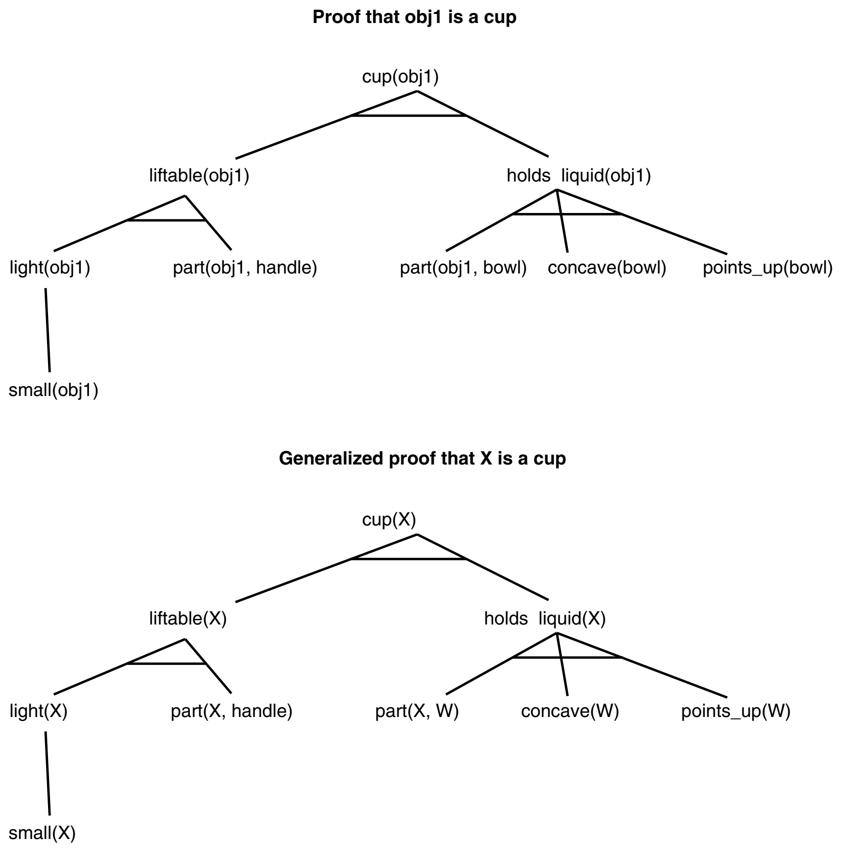


Figure 10.17 Specific and generalized proof that an object, X, is a cup.

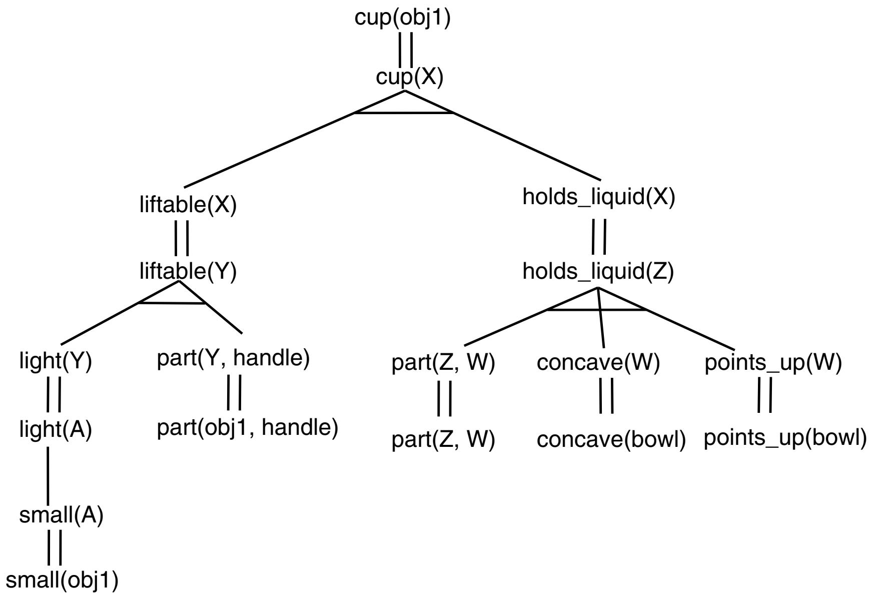


Figure 10.18 An explanation structure of the cup example.

originated in the domain theory rather than the training instance. We have retained it as an essential constraint in the acquired rule.

We may construct a generalized proof tree in a number of ways using a training instance as a guide. Mitchell et al. (1986) accomplish this by first constructing a proof tree that is specific to the training example and subsequently generalizing the proof through a process called *goal regression*. Goal regression matches the generalized goal (in our example, cup(X)) with the root of the proof tree, replacing constants with variables as required for the match. The algorithm applies these substitutions recursively through the tree until all appropriate constants have been generalized. See Mitchell et al. (1986) for a detailed description of this process.

DeJong and Mooney (1986) propose an alternative approach that essentially builds the generalized and the specific trees in parallel. This is accomplished by maintaining a variation of the proof tree consisting of the rules used in proving the goal distinct from the variable substitutions used in the actual proof. This is called an *explanation structure*, as in Figure 10.18, and represents the abstract structure of the proof. The learner maintains two distinct substitution lists for the explanation structure: a list of the specific substitutions required to explain the training example and a list of general substitutions required to explain the generalized goal. It constructs these substitution lists as it builds the explanation structure.

We construct the lists of general and specific substitutions as follows: let s_s and s_g be the lists of specific and general substitutions, respectively. For every match between expressions e_1 and e_2 in the explanation structure, update s_s and s_g according to the following rule:

```
if $e_1$ is in the premise of a domain rule and $e_2$ is the conclusion of a domain rule
then begin
  $T_s$  = the most general unifier of $e_1s_s$ and $e_2s_s$        % unify $e_1$ and $e_2$ under $s_s$
  $s_s$  = $s_sT_s$                                        % update $s_s$ by composing it with $T_s$
  $T_g$  = the most general unifier of $e_1s_g$ and $e_2s_g$        % unify $e_1$ and $e_2$ under $s_g$
  $s_g$  = $s_gT_g$                                        % update $s_g$ by composing it with $T_g$
end
if $e_1$ is in the premise of a domain rule and $e_2$ is a fact in the training instance
then begin                                                   % only update $s_s$
  $T_s$  = the most general unifier of $e_1s_s$ and $e_2s_s$        % unify $e_1$ and $e_2$ under $s_s$
  $s_s$  = $s_s T_s$                                       % update $s_s$ by composing it with $T_s$
end
```

In the example of Figure 10.18:

s_s = {obj1/X, obj1/Y, obj1/A, obj1/Z, bowl/W}

s_g = {X/Y, X/A, X/Z}

Applying these substitutions to the explanation structure of Figure 10.18 gives the specific and general proof trees of Figure 10.17.

Explanation-based learning offers a number of benefits:

1. Training examples often contain irrelevant information, such as the color of the cup in the preceding example. The domain theory allows the learner to select the relevant aspects of the training instance.
2. A given example may allow numerous possible generalizations, most of which are either useless, meaningless, or wrong. EBL forms generalizations that are known to be relevant to specific goals and that are guaranteed to be logically consistent with the domain theory.
3. By using domain knowledge EBL allows learning from a single training instance.
4. Construction of an explanation allows the learner to hypothesize unstated relationships between its goals and its experience, such as deducing a definition of a cup based on its structural properties.

EBL has been applied to a number of learning problems. For instance, Mitchell et al. (1983) discuss the addition of EBL to the LEX algorithm. Suppose that the first positive example of the use of OP1 is in solving the instance $\int 7 x^2 dx$. LEX will make this instance a member of S, the set of maximally specific generalizations. However, a human would immediately recognize that the techniques used in solving this instance do not depend upon the specific values of the coefficient and exponent but will work for any real values, so long as the exponent is not equal to −1. The learner is justified in inferring that OP1 should be applied to any instance of the form $\int r_1 x^{(r_2 \neq -1)} dx$, where r_1 and r_2 are any real numbers. LEX has been extended to use its knowledge of algebra with explanation-based learning to make this type of generalization. Finally, we implement an explanation base learning algorithm in Prolog in the auxiliary material for this book.

10.5.3 EBL and Knowledge-Level Learning

Although it is an elegant formulation of the role of knowledge in learning, EBL raises a number of important questions. One of the more obvious ones concerns the issue of what an explanation-based learner actually learns. Pure EBL can only learn rules that are within the *deductive closure* of its existing theory. This means the learned rules could have been inferred from the knowledge base without using the training instance at all. The sole function of the training instance is to focus the theorem prover on relevant aspects of the problem domain. Consequently, EBL is often viewed as a form of *speed up learning* or knowledge base reformulation; EBL can make a learner work faster, because it does not have to reconstruct the proof tree underlying the new rule. However, because it could always have reconstructed the proof, EBL cannot make the learner do anything new. This distinction has been formalized by Dietterich in his discussion of *knowledge-level learning* (1986).

EBL takes information implicit in a set of rules and makes it explicit. For example, consider the game of chess: a minimal knowledge of the rules of chess, when coupled with an ability to perform unlimited look-ahead on board states, would allow a computer to play extremely well. Unfortunately, chess is too complex for this approach. An explanation-based learner that could master chess strategies would indeed learn something that was, for all practical purposes, new.

EBL also allows us to abandon the requirement that the learner have a complete and correct theory of the domain and focus on techniques for refining incomplete theories within the context of EBL. Here, the learner constructs a partial solution tree. Those branches of the proof that cannot be completed indicate deficiencies in the theory. A number of interesting questions remain to be examined in this area. These include the development of heuristics for reasoning with imperfect theories, credit assignment methodologies, and choosing which of several failed proofs should be repaired.

A further use for explanation-based learning is to integrate it with similarity-based approaches to learning. Again, a number of basic schemes suggest themselves, such as using EBL to refine training data where the theory applies and then passing this partially generalized data on to a similarity-based learner for further generalization. Alternatively, we could use failed explanations as a means of targeting deficiencies in a theory, thereby guiding data collection for a similarity-based learner.

Other issues in EBL research include techniques for reasoning with unsound theories, alternatives to theorem proving as a means of constructing explanations, methods of dealing with noisy or missing training data, and methods of determining which generated rules to save.

10.5.4 Analogical Reasoning

Whereas "pure" EBL is limited to deductive learning, analogies offer a more flexible method of using existing knowledge. Analogical reasoning assumes that if two situations are known to be similar in some respects, it is likely that they will be similar in others. For example, if two houses have similar locations, construction, and condition, then they probably have about the same sales value. Unlike the proofs used in EBL, analogy is not logically sound. In this sense it is like induction. As Russell (1989) and others have observed, analogy is a species of single instance induction: in our house example, we are inducing properties of one house from what is known about another.

As we discussed in our presentation of case-based reasoning (Section 7.3), analogy is very useful for applying existing knowledge to new situations. For example, assume that a student is trying to learn about the behavior of electricity, and assume that the teacher tells her that electricity is analogous to water, with voltage corresponding to pressure, amperage to the amount of flow, and resistance to the capacity of a pipe. Using analogical reasoning, the student may more easily grasp such concepts as Ohm's law.

The standard computational model of analogy defines the *source* of an analogy to be a problem solution, example, or theory that is relatively well understood. The *target* is not completely understood. Analogy constructs a *mapping* between corresponding elements of the target and source. Analogical inferences extend this mapping to new elements of the target domain. Continuing with the "electricity is like water" analogy, if we know that this analogy maps switches onto valves, amperage onto quantity of flow, and voltage onto water pressure, we may reasonably infer that there should be some analogy to the capacity (i.e., the cross-sectional area) of a water pipe; this could lead to an understanding of electrical resistance.

A number of authors have proposed a unifying framework for computational models of analogical reasoning (Hall 1989, Kedar-Cabelli 1988, Wolstencroft 1989). A typical framework consists of the following stages:

1. *Retrieval*. Given a target problem, it is necessary to select a potential source analog. Problems in analogical retrieval include selecting those features of the target and source that increase the likelihood of retrieving a useful source analog and indexing knowledge according to those features. Generally, retrieval establishes the initial elements of an analogical mapping.
2. *Elaboration*. Once the source has been retrieved, it is often necessary to derive additional features and relations of the source. For example, it may be necessary to develop a specific problem-solving trace (or explanation) in the source domain as a basis for analogy with the target.
3. *Mapping and inference*. This stage involves developing the mapping of source attributes into the target domain. This involves both known similarities and analogical inferences.
4. *Justification*. Here we determine that the mapping is indeed valid. This stage may require modification of the mapping.

5. *Learning*. In this stage the acquired knowledge is stored in a form that will be useful in the future.

These stages have been developed in a number of computational models of analogical reasoning. For example, *structure mapping theory* (Falkenhainer 1990, Falkenhainer et al. 1989, Gentner 1983) not only addresses the problem of constructing useful analogies but also provides a plausible model of how humans understand analogies. A central question in the use of analogy is how we may distinguish expressive, deep analogies from more superficial comparisons. Gentner argues that true analogies should emphasize systematic, structural features of a domain over more superficial similarities. For example, the analogy, "the atom is like the solar system" is deeper than "the sunflower is like the sun," because the former captures a whole system of causal relations between orbiting bodies whereas the latter describes superficial similarities such as the fact that both sunflowers and the sun are round and yellow. This property of analogical mapping is called *systematicity.*

Structure mapping formalizes this intuition. Consider the example of the atom/solar system analogy, as in Figure 10.19 as explicated by Gentner (1983). The source domain includes the predicates:

```
yellow(sun)
blue(earth)
hotter-than(sun, earth)
causes(more-massive(sun, earth), attract(sun, earth))
causes(attract(sun, earth), revolves-around(earth, sun))
```

The target domain that the analogy is intended to explain includes

```
more-massive(nucleus, electron)
revolves-around(electron, nucleus)
```

Structure mapping attempts to transfer the causal structure of the source to the target. The mapping is constrained by the following rules:

1. Properties are dropped from the source. Because analogy favors systems of relations, the first stage is to eliminate those predicates that describe superficial properties of the source. Structure mapping formalizes this by eliminating predicates of a single argument (unary predicates) from the source. The rationale for this is that predicates of higher arity, by virtue of describing a relationship between two or more entities, are more likely to capture the systematic relations intended by the analogy. In our example, this eliminates such assertions as yellow(sun) and blue(earth). Note that the source may still contain assertions, such as hotter-than(sun, earth), that are not relevant to the analogy.
2. Relations map unchanged from the source to the target; the arguments to the relations may differ. In our example, such relations as revolves-around and more-massive are the same in both the source and the target. This constraint is

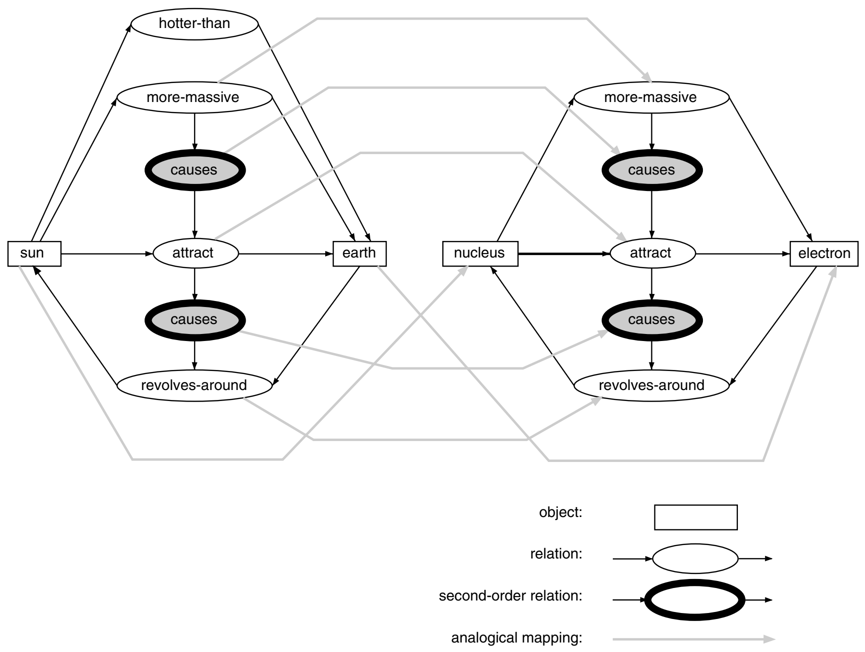


Figure 10.19 An analogical mapping.

used by many theories of analogy and greatly reduces the number of possible mappings. It is also consistent with the heuristic of giving relations preference in the mapping.

3. In constructing the mapping, higher-order relations are preferred as a focus of the mapping. In our example, causes is such a higher-order relation, because it takes other relations as its arguments. This mapping bias is called the *systematicity principle*.

These constraints lead to the mapping:

sun → nucleus
earth → electron

Extending the mapping leads to the inference:

causes(more-massive(nucleus, electron), attract(nucleus, electron))
causes(attract(nucleus, electron), revolves-around(electron, nucleus))

Structure mapping theory has been implemented and tested in a number of domains. Though it remains far from a complete theory of analogy, failing to address such problems as source analog retrieval, it has proven both computationally practical and able to explain many aspects of human analogical reasoning. Finally, as we noted in our presentation of *case-based reasoning,* Section 7.3, there is an essential role for analogy in creating and applying a useful case base.

10.6 Unsupervised Learning

The learning algorithms discussed so far implement forms of *supervised learning*. They assume the existence of a teacher, some fitness measure, or other external method of classifying training instances. *Unsupervised learning* eliminates the teacher and requires that the learners form and evaluate concepts on their own. Science is perhaps the best example of unsupervised learning in humans. Scientists do not have the benefit of a teacher. Instead, they propose hypotheses to explain observations; evaluate their hypotheses using such criteria as simplicity, generality, and elegance; and test hypotheses through experiments of their own design.

10.6.1 Discovery and Unsupervised Learning

AM (Davis and Lenat 1982, Lenat and Brown 1984) is one of the earliest and most successful discovery programs, deriving a number of interesting, even if not original, concepts in mathematics. AM began with the concepts of set theory, operations for creating new knowledge by modifying and combining existing concepts, and a set of heuristics for detecting "interesting" concepts. By searching this space of mathematical concepts, AM discovered the natural numbers along with several important concepts of number theory, such as the existence of prime numbers.

For example, AM discovered the natural numbers by modifying its notion of "bags." A bag is a generalization of a set that allows multiple occurrences of the same element. For example, {a, a, b, c, c} is a bag. By specializing the definition of bag to allow only a single type of element, AM discovered an analogy of the natural numbers. For example, the bag {1, 1, 1, 1} corresponds to the number 4. Union of bags led to the notion of addition: $\{1,1\} \cup \{1,1\} = \{1,1,1,1\}$, or $2 + 2 = 4$. Exploring further modifications of these concepts, AM discovered multiplication as a series of additions. Using a heuristic that defines new operators by inverting existing operators, AM discovered integer division. It found the concept of prime numbers by noting that certain numbers had exactly two divisors (themselves and 1).

On creating a new concept, AM evaluates it according to a number of heuristics, keeping those concepts that prove "interesting". AM determined that prime numbers were interesting based on the frequency with which they occur. In evaluating concepts using this heuristic, AM generates instances of the concept, testing each to see whether the concept holds. If a concept is true of all instances it is a tautology, and AM gives it a low evaluation. Similarly, AM rejects concepts that are true of no instances. If a concept is true of a significant portion of the examples (as is the case with prime numbers), AM evaluates it as interesting and selects it for further modification.

Although AM discovered prime numbers and several other interesting concepts, it failed to progress much beyond elementary number theory. In a later analysis of this work, Lenat and Brown (1984) examine the reasons for the program's success and its limitations. Although Lenat originally believed that AM's heuristics were the prime source of its power, this later evaluation attributed much of the program's success to the language used to represent mathematical concepts. AM represented concepts as recursive structures in a variation of the Lisp programming language. Because of its basis in a well-designed programming language, this representation defined a space that contained a high density of interesting concepts. This was particularly true in the early stages of the search. As exploration continued, the space grew combinatorially, and the percentage of interesting concepts "thinned out." This observation further underscores the relationship between representation and search.

Another reason AM failed to continue the impressive pace of its early discoveries is its inability to "learn to learn." It did not acquire new heuristics as it gained mathematical knowledge; consequently, the quality of its search degraded as its mathematics grew more complex. In this sense, AM never developed a deep understanding of mathematics. Lenat has addressed this problem in later work on a program called EURISKO, which attempts to learn new heuristics (Lenat 1983).

A number of programs have continued to explore the problems of automatic discovery. IL (Sims 1987) applies a variety of learning techniques to mathematical discovery, including methods such as theorem proving and explanation-based learning (Section 10.5). See also the automated invention of integer sequences in Cotton et al. (2000).

BACON (Langley et al. 1986, 1987) has developed computational models of the formation of quantitative scientific laws. For example, using data that related the distances of the planets from the sun and the period of the planets' orbits, BACON "re-discovered" Kepler's laws of planetary motion. By providing a plausible computational model of how humans may have achieved discovery in a variety of domains, BACON has provided a useful tool and methodology for examining the process of human scientific discovery. SCAVENGER (Stubblefield 1995, Stubblefield and Luger 1996) used a variation of the ID3 algorithm to improve its ability to form useful analogies. Shrager and Langley (1990) describe a number of other discovery systems.

Although scientific discovery is an important research area, progress to date has been slight. A more basic, and perhaps more fruitful problem in unsupervised learning, concerns the discovery of categories. Lakoff (1987) suggests that categorization is fundamental to human cognition: higher-level theoretical knowledge depends upon the ability to organize the particulars of our experience into coherent taxonomies. Most of our useful knowledge is about categories of objects, such as cows, rather than about specific individual cows, such as Blossom or Ferdinand. Nordhausen and Langley have emphasized the formation of categories as the basis for a unified theory of scientific discovery (Nordhausen and Langley 1990). In developing explanations of why chemicals react in the ways they do, chemistry built on prior work in classifying compounds into categories such as "acid" and "alkaline."

In the next section, we examine *conceptual clustering*, which is the problem of discovering useful categories in unclassified data.

10.6.2 Conceptual Clustering

The *clustering problem* begins with a collection of unclassified objects and a means for measuring the similarity of objects. The goal is to organize the objects into classes that meet some standard of quality, such as maximizing the similarity of objects in the same class.

Numeric taxonomy is one of the oldest approaches to the clustering problem. Numeric methods rely upon the representation of objects as a collection of features, each of which may have some numeric value. A reasonable similarity metric treats each object (a vector of n feature values) as a point in n-dimensional space. The similarity of two objects is the euclidean distance between them in this space.

Using this similarity metric, a common clustering algorithm builds clusters in a bottom-up fashion. This approach, often called an *agglomerative clustering* strategy, forms categories by:

1. Examining all pairs of objects, selecting the pair with the highest degree of similarity, and making that pair a cluster.
2. Defining the features of the cluster as some function, such as average, of the features of the component members and then replacing the component objects with this cluster definition.
3. Repeating this process on the collection of objects until all objects have been reduced to a single cluster.
4. Many unsupervised learning algorithms can be viewed as performing *maximum liklihood density estimations*, which means finding a distribution from which the data is most likely to have been drawn. An example is the interpretation of a set of phonemes in a natural language application, see Chapter 15.

The result of this algorithm is a binary tree whose leaf nodes are instances and whose internal nodes are clusters of increasing size.

We may extend this algorithm to objects represented as sets of symbolic, rather than numeric, features. The only problem is in measuring the similarity of objects defined using symbolic rather than numeric values. A reasonable approach defines the similarity of two objects as the proportion of features that they have in common. Given the objects

```
object1 = {small, red, rubber, ball}
object2 = {small, blue, rubber, ball}
object3 = {large, black, wooden, ball}
```

this metric would compute the similarity values:

similarity(object1, object2) = $^3/_4$
similarity(object1, object3) = similarity(object2, object3) = $^1/_4$

However, similarity-based clustering algorithms do not adequately capture the underlying role of semantic knowledge in cluster formation. For example, constellations of stars are described both on the basis of their closeness in the sky as well as by way of existing human concepts, such as “the big dipper”.

In defining categories, we cannot give all features equal weight. In any given context, certain of an object’s features are more important than others; simple similarity metrics treat all features equally. Human categories depend upon the goals of the categorization and prior knowledge of the domain much more than on surface similarity. Consider, for example, the classification of whales as mammals instead of fish. Surface similarities cannot account for this classification, which depends upon the wider goals of biological classification and extensive physiological and evolutionary evidence.

Traditional clustering algorithms not only fail to take goals and background knowledge into account, but they also fail to produce meaningful semantic explanations of the resulting categories. These algorithms represent clusters *extensionally*, which means by enumerating all of their members. The algorithms produce no *intensional* definition, or no general rule that defines the semantics of the category and that may be used to classify both known and future members of the category. For example, an extensional definition of the set of people who have served as secretary-general of the United Nations would simply list those individuals. An intensional definition, such as:

{X | X has been elected secretary-general of the United Nations}

would have the added benefits of defining the class semantically and allowing us to recognize future members of the category.

Conceptual clustering addresses these problems by using machine learning techniques to produce general concept definitions and applying background knowledge to the formation of categories. CLUSTER/2 (Michalski and Stepp 1983) is a good example of this approach. It uses background knowledge in the form of biases on the language used to represent categories.

CLUSTER/2 forms k categories by constructing individuals around k *seed* objects. k is a parameter that may be adjusted by the user. CLUSTER/2 evaluates the resulting clusters, selecting new seeds and repeating the process until its quality criteria are met. The algorithm is defined:

1. Select k seeds from the set of observed objects. This may be done randomly or according to some selection function.
2. For each seed, using that seed as a positive instance and all other seeds as negative instances, produce a maximally general definition that covers all of the positive and none of the negative instances. Note that this may lead to multiple classifications of other, nonseed, objects.
3. Classify all objects in the sample according to these descriptions. Replace each maximally general description with a maximally specific description that covers all objects in the category. This decreases likelihood that classes overlap on unseen objects.
4. Classes may still overlap on given objects. CLUSTER/2 includes an algorithm for adjusting overlapping definitions.

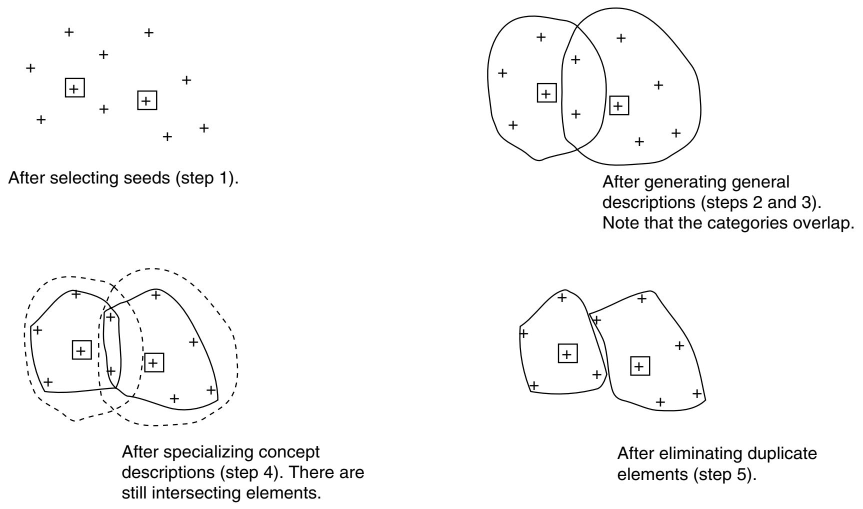

After selecting seeds (step 1).
After generating general descriptions (steps 2 and 3). Note that the categories overlap.
After specializing concept descriptions (step 4). There are still intersecting elements.
After eliminating duplicate elements (step 5).

Figure 10.20 The steps of a CLUSTER/2 run.

5. Using a distance metric, select an element closest to the center of each class. The distance metric could be similar to the similarity metric discussed above.
6. Using these central elements as new seeds, repeat steps 1–5. Stop when clusters are satisfactory. A typical quality metric is the complexity of the general descriptions of classes. For instance, a variation of Occam's Razor might prefer clusters that yield syntactically simple definitions, such as those with a small number of conjuncts.
7. If clusters are unsatisfactory and no improvement occurs over several iterations, select the new seeds closest to the edge of the cluster, rather than those at the center.

Figure 10.20 shows the stages of a CLUSTER/2 execution.

10.6.3 COBWEB and the Structure of Taxonomic Knowledge

Many clustering algorithms, as well as many supervised learning algorithms such as ID3, define categories in terms of necessary and sufficient conditions for membership. These conditions are a set of properties possessed by all members of a category and only by members of the category. Though many categories, such as the set of all United Nations delegates, may be so defined, human categories do not always fit this model. Indeed, human categorization is characterized by greater flexibility and a much richer structure than we have so far examined.

For example, if human categories were indeed defined by necessary and sufficient conditions for membership, we could not distinguish degrees of category membership.

However, psychologists have noted a strong sense of prototypicality in human categorization (Rosch 1978). For instance, we generally think of a robin as a better example of a bird than a chicken; an oak is a more typical example of a tree than a palm (at least in northern latitudes).

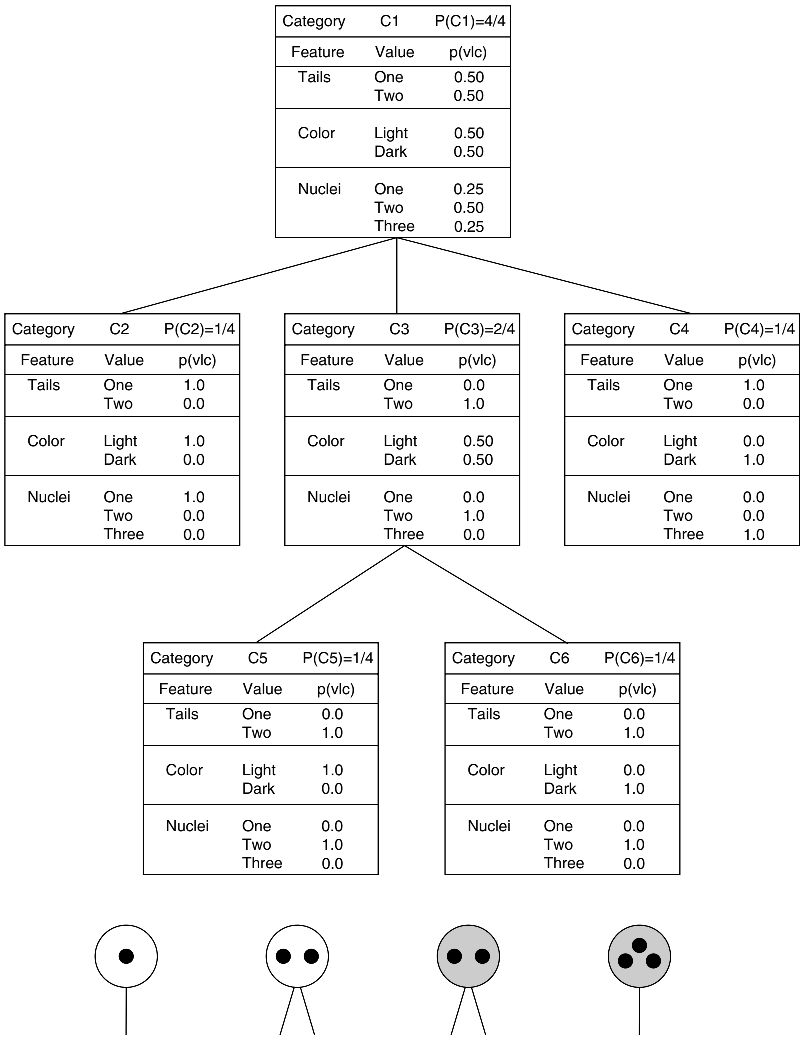
Family resemblance theory (Wittgenstein 1953) supports these notions of prototypicality by arguing that categories are defined by complex systems of similarities between members, rather than by necessary and sufficient conditions for membership. Such categories may not have any properties shared by all of their members. Wittgenstein cites the example of games: not all games require two or more players, such as solitaire; not all games are fun for the players, such as rochambeau; not all games have well-articulated rules, such as children's games of make believe; and not all games involve competition, such as jumping rope. Nonetheless, we consider the category to be well-defined and unambiguous.

Human categories also differ from most formal inheritance hierarchies in that not all levels of human taxonomies are equally important. Psychologists (Rosch 1978) have demonstrated the existence of *base-level* categories. The base-level category is the classification most commonly used in describing objects, the terminology first learned by children, and the level that in some sense captures the most fundamental classification of an object. For example, the category "chair" is more basic than either its generalizations, such as "furniture," or its specializations, such as "office chair." "Car" is more basic than either "sedan" or "vehicle."

Common methods of representing class membership and hierarchies, such as logic, inheritance systems, feature vectors, or decision trees, do not account for these effects. Yet doing so is not only important to cognitive scientists, whose goal is the understanding of human intelligence; it is also valuable to the engineering of useful AI applications. Users evaluate a program in terms of its flexibility, its robustness, and its ability to behave in ways that seem reasonable by human standards. Although we do not require that AI algorithms parallel the architecture of the human mind, any algorithm that proposes to discover categories must meet user expectations as to the structure and behavior of those categories.

COBWEB (Fisher 1987) addresses these issues. Although it is not intended as a model of human cognition, it does account for base-level categorization and degrees of category membership. In addition, COBWEB learns incrementally: it does not require that all instances be present before it begins learning. In many applications, the learner acquires data over time. In these situations, it must construct usable concept descriptions from an initial collection of data and update those descriptions as more data become available. COBWEB also addresses the problem of determining the correct number of clusters. CLUSTER/2 produced a prespecified number of categories. Although the user could vary this number or the algorithm could try different values in an effort to improve categorization, such approaches are not particularly flexible. COBWEB uses global quality metrics to determine the number of clusters, the depth of the hierarchy, and the category membership of new instances.

Unlike the algorithms we have seen so far, COBWEB represents categories probabilistically. Instead of defining category membership as a set of values that must be present for each feature of an object, COBWEB represents the probability with which



Category	C1	P(C1)=4/4
Feature	Value	p(v|c)
Tails	One	0.50
	Two	0.50
Color	Light	0.50
	Dark	0.50
Nuclei	One	0.25
	Two	0.50
	Three	0.25

Category	C2	P(C2)=1/4
Feature	Value	p(v|c)
Tails	One	1.0
	Two	0.0
Color	Light	1.0
	Dark	0.0
Nuclei	One	1.0
	Two	0.0
	Three	0.0

Category	C3	P(C3)=2/4
Feature	Value	p(v|c)
Tails	One	0.0
	Two	1.0
Color	Light	0.50
	Dark	0.50
Nuclei	One	0.0
	Two	1.0
	Three	0.0

Category	C4	P(C4)=1/4
Feature	Value	p(v|c)
Tails	One	1.0
	Two	0.0
Color	Light	0.0
	Dark	1.0
Nuclei	One	0.0
	Two	0.0
	Three	1.0

Category	C5	P(C5)=1/4
Feature	Value	p(v|c)
Tails	One	0.0
	Two	1.0
Color	Light	1.0
	Dark	0.0
Nuclei	One	0.0
	Two	1.0
	Three	0.0

Category	C6	P(C6)=1/4
Feature	Value	p(v|c)
Tails	One	0.0
	Two	1.0
Color	Light	0.0
	Dark	1.0
Nuclei	One	0.0
	Two	1.0
	Three	0.0

Figure 10.21 A COBWEB clustering for four one-celled organisms, adapted from Gennari et al.(1989).

each feature value is present. $p(f_i = v_{ij} \mid c_k)$ is the conditional probability (Section 5.2) with which feature f_i will have value v_{ij}, given that an object is in category c_k.

Figure 10.21 illustrates a COBWEB taxonomy taken from Gennari et al. (1989). In this example, the algorithm has formed a categorization of the four single-cell animals at the bottom of the figure. Each animal is defined by its value for the features: color, and numbers of tails and nuclei. Category C3, for example, has a 1.0 probability of having 2 tails, a 0.5 probability of having light color, and a 1.0 probability of having 2 nuclei.

As the figure illustrates, each category in the hierarchy includes probabilities of occurrence for all values of all features. This is essential to both categorizing new instances and modifying the category structure to better fit new instances. Indeed, as an incremental algorithm, COBWEB does not separate these actions. When given a new instance, COBWEB considers the overall quality of either placing the instance in an existing category or modifying the hierarchy to accommodate the instance. The criterion COBWEB uses for evaluating the quality of a classification is called *category utility* (Gluck and Corter 1985). Category utility was developed in research on human categorization. It accounts for base-level effects and other aspects of human category structure.

Category utility attempts to maximize both the probability that two objects in the same category have values in common and the probability that objects in different categories will have different property values. Category utility is defined:

$$\sum_k \sum_i \sum_j p(f_i = v_{ij}) p(f_i = v_{ij} | c_k) p(c_k | f_i = v_{ij})$$

This sum is taken across all categories, c_k, all features, f_i, and all feature values, v_{ij}. $p(f_i = v_{ij} \mid c_k)$, called *predictability*, is the probability that an object has value v_{ij} for feature f_i given that the object belongs to category c_k. The higher this probability, the more likely two objects in a category share the same feature values. $p(c_k | f_i = v_{ij})$, called *predictiveness*, is the probability with which an object belongs to category c_k given that it has value v_{ij} for feature f_i. The greater this probability, the less likely objects not in the category will have those feature values. $p(f_i = v_{ij})$ serves as a weight, assuring that frequently occurring feature values will exert a stronger influence on the evaluation. By combining these values, high category utility measures indicate a high likelihood that objects in the same category will share properties, while decreasing the likelihood of objects in different categories having properties in common.

The COBWEB algorithm is defined:

```
cobweb(Node, Instance)
begin
   if Node is a leaf
      then begin
         create two children of Node, $L_1$ and $L_2$;
         set the probabilities of $L_1$ to those of Node;
         initialize the probabilities for $L_2$ to those of Instance;
         add Instance to Node, updating Node s probabilities;
      end
      else begin
         add Instance to Node, updating Node s probabilities;
         for each child, C, of Node, compute the category utility of the clustering
            achieved by placing Instance in C;
         let $S_1$ be the score for the best categorization, $C_1$;
         let $S_2$ be the score for the second best categorization, $C_2$;
         let $S_3$ be the score for placing instance in a new category;
         let $S_4$ be the score for merging $C_1$ and $C_2$ into one category;
         let $S_5$ be the score for splitting $C_1$ (replacing it with its child categories)
      end
```

```
        If $S_1$ is the best score
          then cobweb($C_1$, Instance)                              % place the instance in $C_1$
          else if $S_3$ is the best score
            then initialize the new category s probabilities to those of Instance
            else if $S_4$ is the best score
              then begin
                let $C_m$ be the result of merging $C_1$ and $C_2$;
                cobweb($C_m$, Instance)
              end
              else if $S_5$ is the best score
                then begin
                  split $C_1$;
                  cobweb(Node, Instance)
              end;
    end
```

COBWEB performs a hill-climbing search of the space of possible taxonomies using category utility to evaluate and select possible categorizations. It initializes the taxonomy to a single category whose features are those of the first instance. For each subsequent instance, the algorithm begins with the root category and moves through the tree. At each level it uses category utility to evaluate the taxonomies resulting from:

1. Placing the instance in the best existing category.
2. Adding a new category containing only the instance.
3. Merging of two existing categories into one new one and adding the instance to that category.
4. Splitting of an existing category into two and placing the instance in the best new resulting category.

Figure 10.22 illustrates the processes of merging and splitting nodes. To merge two nodes, the algorithm creates a new node and makes the existing nodes children of that node. It computes the probabilities for the new node by combining the probabilities for the children. Splitting replaces a node with its children.

This algorithm is efficient and produces taxonomies with a reasonable number of classes. Because it allows probabilistic membership, its categories are flexible and robust. In addition, it has demonstrated base-level category effects and, through its notion of partial category matching, supports notions of prototypicality and degree of membership. Instead of relying on two-valued logic, COBWEB, like fuzzy logic, views the “vagueness” of category membership as a necessary component for learning and reasoning in a flexible and intelligent fashion.

We next present reinforcement learning which, like the classifier systems of Section 11.2, interprets feedback from an environment to learn optimal sets of condition/response relationships for problem solving within that environment.

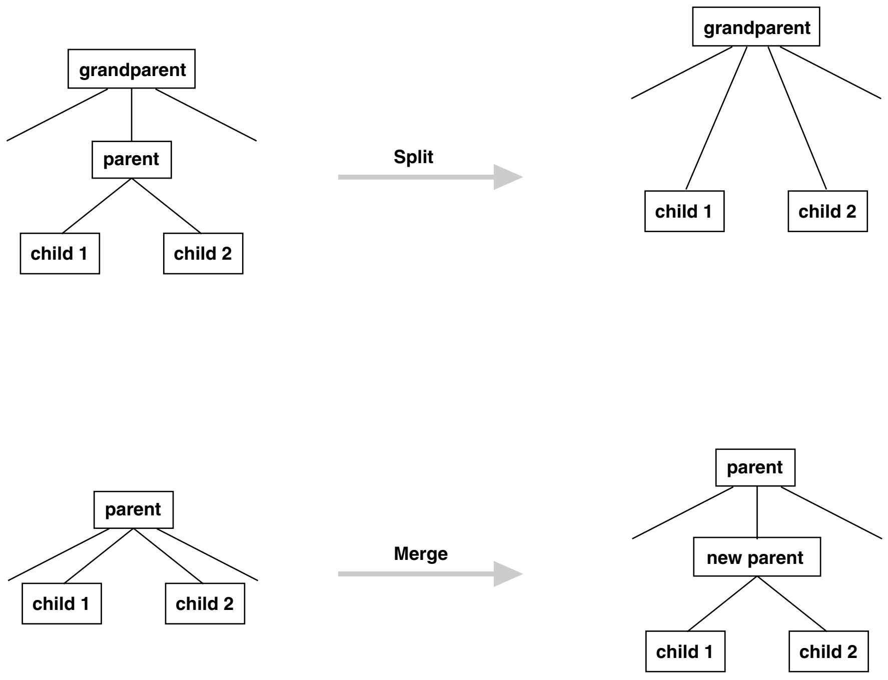


Figure 10.22 Merging and splitting of nodes.

10.7 Reinforcement Learning

We humans (usually!) learn from interacting with our environment. A moment of thought, however, reminds us that feedback from our actions in the world is not always immediate and straightforward. In human relationships, for example, it often takes some time to appreciate fully the results of our actions. Interaction with the world demonstrates for us cause and effect (Pearl 2000), the consequences of our actions, and even how to achieve complex goals. As intelligent agents, we make policies for working in and through our world. "The world" *is* a teacher, but her lessons are often subtle and sometimes hard won!

10.7.1 The Components of Reinforcement Learning

In reinforcement learning, we design computational algorithms for transforming world situations into actions in a manner that maximizes a reward measure. Our agent is not told directly what to do or which action to take; rather, the agent discovers through exploration which actions offer the most reward. Agents' actions affect not just immediate reward, but also impact subsequent actions and eventual rewards. These two features, trial-and-error search and delayed reinforcement, are the two most important characteristics of reinforcement learning. Consequently, reinforcement learning is a more general methodology than the learning seen earlier in this chapter.

Reinforcement learning is not defined by particular learning methods, but by actions within and responses from an environment. Any learning method creating this interaction is an acceptable reinforcement learning method. Reinforcement learning is also not supervised, as seen throughout our chapters on machine learning. In supervised learning, a "teacher" uses examples to directly instruct or train the learner. In reinforcement learning, the learning agent itself, through trial, error, and feedback, learns an optimal policy for accomplishing goals within its environment. (See classifier learning , Section 11.2.)

Another trade-off that the reinforcement learner must consider is between using only what it presently knows and further exploring its world. To optimize its reward possibilities, the agent must not just do what it already knows, but also explore those parts of its world that are still unknown. Exploration allows the agent to (possibly) make better selections in the future; thus obviously, the agent that either always or never explores will usually fail. The agent must explore a variety of options and at the same time favor those that appear to be best. On tasks with stochastic parameters, exploratory actions must often be made multiple times to gain reliable estimates of rewards. (See Chapter 13 for details on stochastic aspects of reinforcement learning.)

Many of the problem-solving algorithms presented earlier in this book, including planners, decision makers, and search algorithms can be viewed in the context of reinforcement learning. For example, we can create a plan with a teleo-reactive controller (Section 7.4) and then evaluate its success with a reinforcement learning algorithm. In fact, the DYNA-Q reinforcement algorithm (Sutton 1990, 1991) integrates model learning with planning and acting. Thus reinforcement learning offers a method for evaluating both plans and models and their utility for accomplishing tasks in complex environments.

We now introduce some terminology for reinforcement learning:

t is a discrete time step in the problem solving process
s_t is the problem state at t, dependent on s_{t-1} and a_{t-1}
a_t is the action at t, dependent on s_t
r_t is the reward at t, dependent on s_{t-1} and a_{t-1}
π is a policy for taking an action in a state. Thus π is a mapping from states to actions
π^* is the optimal policy
V maps a state to its value. Thus, $V^\pi(s)$ is the value of state s under policy π

In Section 10.7.2, *temporal difference learning* learns a V for each s with static π.

There are four components of reinforcement learning, a *policy* π, a *reward function* r, a *value mapping* V, and quite often, a *model* of the environment. The *policy* defines the learning agent's choices and method of action at any given time. Thus the policy could be represented by a set of production rules or a simple lookup table. The policy for a particular situation, as just noted, could also be the result of extensive search, consulting a model, or of a planning process. It could also be stochastic. The policy is the critical component of the learning agent in that it alone is sufficient to produce behavior at any time.

The *reward function* r_t defines the state/goal relationships of the problem at time t. It maps each action, or more precisely each state-response pair, into a reward measure, indicating the desirability of that action for achieving the goal. The agent in reinforcement learning has the task of maximizing the total reward it receives in accomplishing its task.

The *value function* V is a property of each state of the environment indicating the reward the system can expect for actions continuing on from that state. While the reward function measures the immediate desirability of state-response pairs, the value function indicates the long-term desirability of a state of the environment. A state gets its value from both its own intrinsic quality as well as from the quality of states likely to follow from it, i.e., the reward of being in those states. For example, a state/action might have a low immediate reward, but have a high value because it is usually followed by other states that do yield a high reward. A low value could also indicate states that are not associated with eventually successful solution paths.

Without a reward function there are no values and the only purpose of estimating values is to gain more rewards. In making decisions, however, it is values that most interest us, as values indicate states and combinations of states that bring highest rewards. It is much harder to determine values for states, however, than to determine rewards. Rewards are given directly by the environment, while values are estimated and then re-estimated from the successes and failures over time. In fact, the most critical as well as most difficult aspect of reinforcement learning is creating a method for efficiently determining values. We demonstrate one method, a temporal difference learning rule, in Section 10.7.2.

A final – and optional – element for reinforcement learning is the *model* of the environment. A model is a mechanism for capturing aspects of the behavior of the environment. As we saw in Section 7.3, models can be used not only for determining faults as in diagnostic reasoning but also as part of determining a plan of action. Models let us evaluate possible future actions without actually experiencing them. Model-based planning is a recent addition to the reinforcement learning paradigm, as early systems tended to create reward and value parameters based on the pure trial and error actions of an agent.

10.7.2 An Example: Tic-Tac-Toe Revisited

We next demonstrate a reinforcement learning algorithm for tic-tac-toe, a problem we have already considered (Chapter 4), and one dealt with in the reinforcement learning literature by Sutton and Barto (1998). It is important to compare and contrast the reinforcement learning approach with other solution methods, for example, mini-max.

As a reminder, tic-tac-toe is a two-person game played on a 3x3 grid, as in Figure II.5. The players, X and O, alternate putting their marks on the grid, with the first player that gets three marks in a row, either horizontal, vertical, or diagonal, the winner. As the reader is aware, when this game is played using perfect information and backed up values, Section 4.3, it is always a draw. With reinforcement learning we will be able to do something much more interesting, however. We will show how we can capture the performance of an imperfect opponent, and create a policy that allows us to maximize our advantage over this opponent. Our policy can also evolve as our opponent improves her game, and with the use of a model we will be able to generate forks and other attacking moves!

First, we must set up a table of numbers, one for each possible state of the game. These numbers, that state's *value*, will reflect the current estimate of the probability of

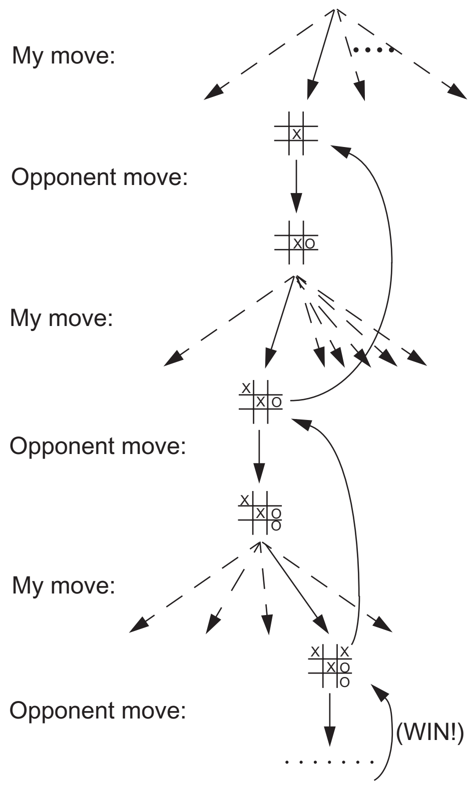


Figure 10.23 A sequence of tic-tac-toe for moves. Dashed arrows indicate possible move choices, down solid arrows indicate selected moves, up solid arrows indicate reward, when reward function changes state s value.

winning from that state. This will support a policy for a strictly winning strategy, i.e., either a win by our opponent or a drawn game will be considered a loss for us. This draw-as-loss approach allows us to set up a policy focused on winning, and is different from our perfect information win-lose-draw model of Section 4.3. This is an important difference, actually; we are attempting to capture the skill of an actual opponent and not the perfect information of some idealized opponent. Thus, we will initialize our value table with a 1 for each win position for us, a 0 for each loss or drawn board, and a 0.5 everywhere else, reflecting the initial guess that we have a 50 per cent chance of winning from those states.

We now play many games against this opponent. For simplicity, let us assume that we are the X and our opponent 0. Figure 10.23 reflects a sequence of possible moves, both those considered and those chosen, within a game situation. To generate a move, we first consider each state that is a legal move from our present state, that is, any open state that we can possibly claim with our X. We look up the current value measure for that state kept in our table. Most of the time we can make a greedy move, that is, taking the state that has the best value function. Occasionally, we will want to make an exploratory move and select randomly from the other states. These exploratory moves are to consider alternatives we might not see within the game situation, expanding possible value optimizations.

While we are playing, we change the *value functions* of each of the states we have selected. We attempt to make their latest values better reflect the probability of their being on a winning path. We earlier called this the *reward function* for a state. To do this we back up the value of a state we have selected as a function of the value of the next state we choose. As can be seen with the "up arrows" of Figure 10.23, this back-up action skips the choice of our opponent, and yet it does reflect the set of values that she has directed us towards for our next choice of a state. Thus, the current value of an earlier state we choose is adjusted to better reflect the value of the later state (and ultimately, of course, of the winning or losing value). We usually do this by moving the previous state some fraction of the value difference between itself and the newer state we have selected. This fractional measure, called the *step-size parameter*, is reflected by the multiplier c in the equation:

$$V(s_n) = V(s_n) + c(V(s_{n+1}) - V(s_n))$$

In this equation, s_n represents the state chosen at time n and s_{n+1} the state chosen at time $n+1$. This update equation is an example of a *temporal difference learning rule*, since its changes are a function of the difference, $V(s_{n+1}) - V(s_n)$, between value estimates at two different times, n and $n+1$. We will discuss these learning rules further in the following section.

The temporal difference rule performs quite well for tic-tac-toe. We want to reduce the step size parameter c over time, so that as the system learns, successively smaller adjustments are made to the state values. This will guarantee convergence of the value functions for each state to the probability of winning, given our opponent. Also, except for periodic exploratory moves, the choices made will in fact be the optimal moves, that is, the optimal policy, against this opponent. What is even more interesting, however, is the fact that if the step size never really gets to zero this policy will continually change to reflect any changes/improvements in the opponent's play.

Our tic-tac-toe example illustrates many of the important features of reinforcement learning. First, there is learning while interacting with the environment, here our opponent. Second, there is an explicit goal (reflected in a number of goal states) and optimal behavior requires planning and look ahead that makes allowance for delayed effects of particular moves. For example, the reinforcement learning algorithm in effect sets up multi move traps for the naive opponent. It is an important feature of reinforcement learning that the effects of look ahead and planning can be in fact achieved, without either an explicit model of the opponent or through extended search.

In our tic-tac-toe example, learning began with no prior knowledge beyond the game's rules. (We simply initialized all non-terminal states to 0.5.) Reinforcement learning certainly does not require this "blank slate" view. Any prior information available can be built into the initial state values. It is also possible to deal with states where there is no information available. Finally, if a model of a situation is available, the resulting model based information can be used for the state values. But it is important to remember that reinforcement learning can be applied in either situation: no model is required, but models can be used if they are available or if they can be learned.

In the tic-tac-toe example, the reward was amortized over each state-action decision. Our agent was myopic, concerned with maximizing only immediate rewards. Indeed, if we use deeper lookahead with reinforcement learning, we will need to measure the *discounted return* of an eventual reward. We let the discount rate γ represent the present value of a future reward: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. This discounting of the reward measure is important in using the dynamic programming approach to reinforcement learning presented in the next section.

Tic-tac-toe was an example of a two-person game. Reinforcement learning can also be used for situations where there are no opponents, just feedback from an environment. The tic-tac-toe example also had a finite (and actually fairly small) state space. Reinforcement learning can also be employed when the state space is very large, or even infinite. In the later case, state values are generated only when that state is encountered and used in a solution. Tesauro (1995) for example, used the temporal difference rule we just described, built into a neural network, to learn to play backgammon. Even though the estimated size of the backgammon state space is 10^{20} states, Tesauro's program plays at the level of the best human players.

10.7.3 Inference Algorithms and Applications for Reinforcement Learning

According to Sutton and Barto (1998) there are three different families of reinforcement learning inference algorithms: *temporal difference* learning, *dynamic programming*, and *Monte Carlo* methods. These three form a basis for virtually all current approaches to reinforcement learning. Temporal difference methods learn from sampled trajectories and back up values from state to state. We saw an example of temporal difference learning with tic-tac-toe in the previous section.

Dynamic programming methods compute value functions by backing up values from successor states to predecessor states. Dynamic programming methods systematically update one state after another, based on a model of the next state distribution. The dynamic programming approach is built on the fact that for any policy π and any state s the following recursive consistency equation holds:

$$V^{\pi}(s) = \sum_{a} \pi(a \mid s) * \sum_{s} \pi(s \rightarrow s \ \mid a) * (R^{a}(s \rightarrow s\ ) + \gamma(V^{\pi}(s\ ))$$

$\pi(a \mid s)$ is the probability of action a given state s under stochastic policy π. $\pi(s \rightarrow s \ \mid a)$ is the probability of s going to s under action a. This is the Bellman (1957) equation for V^{π}. It expresses a relationship between the value of a state and the recursively calculated values of its successor states. (See dynamic programming in Section 4.1.) In Figure 10.24a we present the first step calculation, where, from state s, we look forward to three possible successor states. With policy π, action a has probability $\pi(a \mid s)$ of occurring. From each of these three states the environment could respond with one of several states, say s with reward r. The Bellman equation averages over all these possibilities, weighting each by its probability of happening. It states that the value of the start state s must equal the discounted, γ, value of the expected next states plus the reward generated along the path.

Classical dynamic programming models are of limited utility because of their assumption of a perfect model. If n and m denote the number of states and actions a dynamic programming method is guaranteed to find an optimal policy in polynomial time, even though the total number of deterministic policies is n^m. In this sense dynamic programming is exponentially faster than any direct search in policy space could be, because direct search would have to evaluate each policy exhaustively to give the same guarantee.

Monte Carlo methods do not require a complete model. Instead they sample the entire trajectories of states to update the value function based on the episodes' final outcomes. Monte Carlo methods do require experience, that is, sample sequences of states, actions, and rewards from on-line or simulated interactions with the environment. On-line experience is interesting because it requires no prior knowledge of the environment, yet can still be optimal. Learning from simulated experience is also powerful. A model is required, but it can be generative rather than analytical, that is, a model able to generate trajectories but not able to calculate explicit probabilities. Thus, it need not produce the complete probability distributions of all possible transitions that are required in dynamic programming.

Thus, Monte Carlo methods solve the reinforcement learning problem by averaging sample returns. To ensure well defined returns, Monte Carlo methods are defined only for full episodes, that is, all episodes must eventually terminate. Furthermore, it is only on completion of an episode that the value estimates and policies are changed. Thus, Monte Carlo methods are incremental in an episode by episode sense and not step by step. The term "Monte Carlo" is often used more broadly for any estimation method whose operation involves a significant random component. Here it is used specifically for methods based on averaging complete returns. We consider Monte Carlo reinforcement methods again in Chapter 13.

There are other methods used for reinforcement learning, the most important being Q-learning (Watkins 1989), a variant of the temporal difference approach. In Q-learning, Q is a function of state-action pairs to learned values. For all states and actions:

$$Q: (\text{state x action}) \rightarrow \text{value}$$

For one step Q-learning:

$$Q(s_t,a_t) \leftarrow (1 - c) * Q(s_t,a_t) + c * [r_{t+1} + \gamma * \max_a Q(s_{t+1},a) - Q(s_t,a_t)]$$

where both c and γ are ≤ 1 and r_{t+1} is the reward at s_{t+1}. We can visualize the Q approach in Figure 10.24b, and contrast it with Figure 10.24a, where the start node is a state-action situation. This backup rule updates state-action pairs, so that the top state of Figure 10.24b, the root of the backup, is an action node coupled with the state that produced it.

In Q-learning, the backup is from action nodes, maximizing over all the actions possible from the next states, along with their rewards. In full recursively defined Q-learning, the bottom nodes of the backup tree are all terminal nodes reachable by a sequence of actions starting from the root node, together with the rewards of these successor actions. On-line Q-learning, expanding forward from possible actions, does not require building a full world model. Q-learning can also be performed off-line. As one can see, Q-learning is

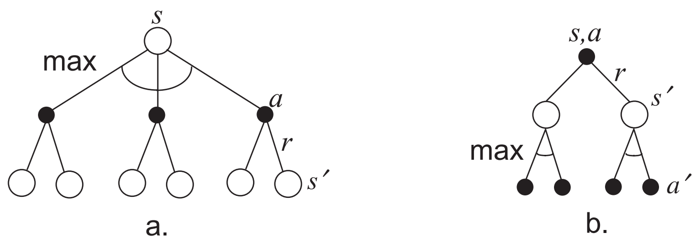


Figure 10.24 Backup diagrams for (a) V* and (b) Q*, adapted from Sutton and Barto (1998).

a kind of temporal difference approach. Further details may be found in Watkins (1989) and Sutton and Barto (1998). Q-learning is discussed again in Chapter 13.

There are now a number of significant problems solved with reinforcement learning, including Backgammon (Tesauro 1994, 1995). Sutton and Barto (1998) also analyze Samuel's checker program, Section 4.3, from the reinforcement learning viewpoint. They also discuss the reinforcement learning approach to the *acrobat*, *elevator dispatching*, *dynamic channel allocation*, *job shop scheduling*, and other classic problem areas (Sutton and Barto 1998).

10.8 Epilogue and References

Machine learning is one of the most exciting subfields in artificial intelligence, addressing a problem that is central to intelligent behavior and raising a number of questions about knowledge representation, search, and even the basic assumptions of AI itself. Surveys of machine learning include Pat Langley's (1995) *Elements of Machine Learning*, Tom Mitchell's (1997) book *Machine Learning,* and Anthony Martin's (1997) *Computational Learning Theory: An Introduction*. *An Introduction to Machine Learning* by Nils Nilsson http://robotics.stanford.edu/people/nilsson/mlbook.html. See also Section 16.2.

An early surveys of learning include *Machine Learning: An Artificial Intelligence Approach* (Kodratoff and Michalski 1990, Michalski et al. 1983, 1986). *Readings in Machine Learning* (Shavlik and Dietterich 1990) collects papers in the field, back as far as 1958. By placing all this research in a single volume, the editors have provided a very valuable service to both researchers and those seeking an introduction to the field. Inductive learning is presented by Vere (1975, 1978) and Dietterich and Michalski (1981, 1986). *Production System Models of Learning and Development* (Klahr et al. 1987) collects a number of papers in machine learning, including work (SOAR) reflecting a more cognitive science approach to learning.

Computer Systems That Learn (Weiss and Kulikowski 1991) is an introductory survey of the whole field, including treatments of neural networks, statistical methods, and machine learning techniques. Readers interested in a deeper discussion of analogical reasoning should examine Carbonell (1983, 1986), Holyoak (1985), Kedar-Cabelli (1988), and Thagard (1988). For those interested in discovery and theory formation, see *Scientific Discovery: Computational Explorations of the Creative Processes* (Langley et al. 1987) and *Computational Models of Scientific Discovery and Theory Formation* (Shrager and Langley 1990). *Concept Formation: Knowledge and Experience in Unsupervised Learning* (Fisher et al. 1991) presents a number of papers on clustering, concept formation, and other forms of unsupervised learning.

ID3 has a long history within the machine learning community. *EPAM*, the *Elementary Perceiver And Memorizer* in (Feigenbaum and Feldman 1963), used a type of decision tree, called a *discrimination net*, to organize sequences of nonsense syllables. Quinlan was the first to use information theory to generate children in the decision tree. Quinlan (1993) and others extended ID3 to C4.5, and address issues such as noise and continuous attributes in data (Quinlan 1996, Auer et al. 1995). Stubblefield and Luger (1996) have applied ID3 to the problem of improving the retrieval of sources in an analogical reasoning system.

Michie (1961) and Samuel (1959) offer early examples of reinforcement learning. Sutton and Barto's (1998) *Reinforcement Learning* was the source of much of our presentation on this topic. We recommend Watkins (1989) thesis for a more detailed presentation of Q-learning, Sutton's (1988) original paper for analysis of temporal difference learning, and Bertsekas and Tsitsiklis (1996), *Neuro-Dynamic Programming*, for a more formal presentation of all reinforcement learning algorithms. We discuss reinforcemnt learning again in the context of probabilistic learning, Chapter 13.

Machine Learning is the primary journal of the field. Other sources of current research include the yearly proceedings of the *International Conference on Machine Learning* and the *European Conference on Machine Learning* as well as the proceedings of the *American Association of Artificial Intelligence* (now *Association for the Advancement of Artificial Intelligence*) conference as well as proceedings of the *International Joint Conference on Artificial Intelligence*. for the latest update on topics in machine Learning.

We present connectionist learning in Chapter 11, social and emergent approaches to learning in Chapter 12, and dynammic and probabilistic learning in Chapter 13. We discuss inductive bias, generalization, and other limitations on learning in Section 16.2.

10.9 Exercises

1. Consider the behavior of Winston's concept learning program when learning the concept "step," where a step consists of a short box and a tall box placed in contact with each other, as in Figure 10.25. Create semantic net representations of three or four examples and near misses and show the development of the concept.

2. The run of the candidate elimination algorithm shown in Figure 10.9 does not show candidate concepts that were produced but eliminated because they were either overly general, overly specific, or subsumed by some other concept. Re-do the execution trace, showing these concepts and the reasons each was eliminated.

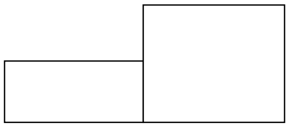


Figure 10.25 A step.

3. Build the version space search algorithms in Prolog, or the language of your choice. If you use Prolog or Java, see hints for version space search in the auxiliary materials.
4. Using the information theoretic selection function of Section 10.4.3, show in detail how ID3 constructs the tree of Figure 10.14 from examples in Table 10.1. Be sure to show the calculations used in computing the information gain for each test and the resulting test selections.
5. Using Shannon's formula, show whether or not a message about the outcome of a spin of a roulette wheel has more information than one about the outcome of a coin toss. What if the roulette wheel message is "not 00"?
6. Develop a simple table of examples in some domain, such as classifying animals by species, and trace the construction of a decision tree by the ID3 algorithm.
7. Implement ID3 in a language of your choice and run it on the credit history example from the text. If you use LISP, consider the algorithms and data structures developed in Section 15.13 for suggestions.
8. Discuss problems that can arise from using continuous attributes in data, such as a monetary cost, dollars and cents, or the height, a real number, of an entity. Suggest some method for addressing this problem of continuous data.
9. Other problems of ID3 are bad or missing data. Data is bad if one set of attributes has two different outcomes. Data is missing if part of the attribute is not present, perhaps because it was too expensive to obtain. How might these issues be dealt with in development of ID3 algorithms?
10. From Quinlan (1993) obtain the C4.5 decision tree algorithm and test it on a data set.There are complete programs and data sets for C4.5 available from this reference.
11. Develop a domain theory for explanation-based learning in some problem area of your choice. Trace the behavior of an explanation-based learner in applying this theory to several training instances.

12) Develop an explanation-based learning algorithm in the language of your choice. If you use Prolog, consider the algorithms developed in the auxiliary material.

13. Consider the tic-tac-toe example of Section 10.7.2. Implement the temporal difference learning algorithm in the language of your choice. If you designed the algorithm to take into account problem symmetries, what do you expect to happen? How might this limit your solution?
14. What happens if the temporal difference algorithm of Problem 13 plays tic-tac-toe against itself?

15. Analyze Samuel's checker playing program from a reinforcement learning perspective. Sutton and Barto (1998, Section 11.2) offer suggestions in this analysis.

16. Can you analyze the inverted pendulum problem, Figure 8.8, presented in Section 8.2.2 from a reinforcement learning perspective? Build some simple reward measures and use the temporal difference algorithm in your analysis.

17. Another problem type excellent for reinforcement learning is the so-called gridworld. We present a simple 4 x 4 gridworld in Figure 10.26. The two greyed corners are the desired terminal states for the agent. From all other states, agent movement is either up, down, left, or right. The agent cannot move off the grid: attempting to, leaves the state unchanged. The reward for all transitions, except to the terminal states is −1. Work through a sequence of grids that produce a solution based on the temporal difference algorithm presented in Section 10.7.2. See Chapter 13 for more discussion of the gridworld problem.

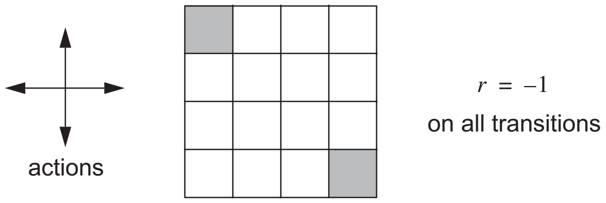


Figure 10.26 An example of a 4 x 4 grid world, adapted from Sutton and Barto (1998).