tion 6.5.2) comes directly from Knerr, Personnaz, and Dreyfus (1990); the method for splitting nodes comes from Hanson (1990), while the cascade correlation algorithm is due to Fahlman and Lebiere (1990). Platt (1991) describes one method for constructing networks of spherical threshold units.

Research on learning in Bayesian networks has been a relatively new development. Cooper and Herskovits (1992) and Heckerman (1995) review techniques for determining the probabilities in such networks from training data, but recent work has focused on inducing the structure of Bayesian networks. Cooper and Herskovits' (1992) K2 uses a greedy algorithm to determine network structure, and others report extensions to this approach (Heckerman, Geiger, & Chickering, 1994; Provan & Singh, 1995). Glymour, Scheines, Spirtes, and Kelly (1987) describe a different approach that includes the ability to posit latent or unobserved variables, whereas Connolly (1993) employs a clustering scheme to infer new latent terms.

CHAPTER 7 The Formation of Concept Hierarchies

In the previous chapter we examined inference networks as an organization of learned knowledge. However, we noted in Chapter 1 that *concept hierarchies* provide an alternative framework for memory organization, and a considerable amount of machine learning research has taken this approach. As we will see, concept hierarchies differ from inference networks not only in their structure, but also in the algorithms that use and construct them.

Much of the work in this area has assumed logical concept descriptions, so our examples will focus on such representations. However, we will also examine some less common methods for organizing threshold and competitive concepts into concept hierarchies. In addition, most research has dealt with nonincremental approaches that construct hierarchies in a 'divisive' manner, from the top down, but we will also consider some incremental variants and some techniques that operate in an 'agglomerative' fashion, constructing hierarchies from the bottom up. As usual, we start by discussing the data structures and performance methods associated with the approach.

7.1 General issues concerning concept hierarchies

Most concept hierarchies assume that instances are described as a conjunction of attribute-value pairs; as in previous chapters, we will emphasize such representations in our examples, even though we will see in Chapter 8 that one can extend the framework to handle relational structures. There is greater variety in the nature of the concept descriptions that hierarchies organize; we will find that one can embed any of the knowledge representations presented in Chapters 2 through 4 within these larger structures. Here we focus on the organization that concept hierarchies impose on memory, on the manner in which a performance element uses these structures, and on the task of acquiring them.

7.1.1 The structure of concept hierarchies

Like an inference network, a concept hierarchy is composed of nodes and links, but with quite different semantics. Each node represents a separate concept, typically with its own associated intensional definition. The links connecting a node to its children specify an 'IS-A' or 'subset' relation, indicating that the parent's extension is a superset of each child's extension. Typically, a node covers all of the instances covered by the union of its descendents, making the concept hierarchy a subgraph of the partial ordering by generality that we discussed in Chapter 2.

We will sometimes distinguish between two types of intensional descriptions stored at nodes. A concept hierarchy uses *predictive* features or descriptions to sort new instances downward through memory. One can view these as 'tests' that discriminate among the concepts stored at each level, although such tests need not be logical in nature. In contrast, a concept hierarchy uses *predictable* features or descriptions to make inferences about aspects of new instances. Some frameworks combine these two aspects of intensional descriptions into a single structure, whereas others make a clear distinction.

For example, consider the simplest and most widely used form of concept hierarchy – the *univariate decision tree*. This organization divides instances into mutually exclusive sets at each level, splitting on the values of a single predictive feature (thus the term *univariate*). As a result, each internal node contains a very simple description for use in sorting. Only terminal nodes include a predictive feature, which specifies the class to predict upon reaching that node. A closely related organizational structure, the *discrimination network*, differs by including multiple predictable features at each terminal node, but the predictive descriptions at internal nodes are typically univariate.

In contrast, other types of concept hierarchies incorporate many predictive features into their internal nodes, in some cases including all available attributes. Decision trees of this sort are termed *multivariate*,

in that each 'test' takes many variables into account.[1] Some memory structures contain a single summary description at each node and treat each feature as both predictive and predictable. For example, one such scheme stores a probabilistic description at each node, which summarizes all instances below it in the hierarchy. In any case, there exist many variations on the hierarchical organization of conceptual knowledge.

7.1.2 The use of concept hierarchies

We have hinted that the classification process involves sorting an instance downward through the hierarchy. At each level, it uses the predictive features on the alternative nodes to select one to expand, then recurses to the next level. This continues until reaching a terminal node or otherwise deciding the sorting process has gone deep enough. At this point, the process uses the predictable features associated with the current node to infer attributes missing from the instance, such as the class name. We have described a greedy sorting scheme, but one can use other techniques as well, including best-first search and even parallel methods that pass the instance down multiple branches at a time.

The details of sorting and prediction depend on the nature of the intensional descriptions stored with the nodes. As we will emphasize repeatedly, concept hierarchies can use any of the description languages we have considered in previous chapters. Thus, they use the interpreters associated with a given concept representation to make sorting decisions. For example, many hierarchies employ logical descriptions and thus use an all-or-none match at each level. Others incorporate threshold units for each node and thus use partial-match schemes. Yet others have instance-based or probabilistic representations of concepts, and so use a best-match procedure to sort instances through memory.

7.1.3 The task of forming concept hierarchies

With issues of representation and performance clearly specified, we can now state the task of constructing concept hierarchies:

- *Given*: A set of training instances and (possibly) associated classes;
- *Find*: A concept hierarchy that, to the extent possible, makes accurate predictions about novel test instances.

1. The numerical taxonomy literature refers to *monothetic* and *polythetic* trees, rather than univariate and multivariate ones.

As with other induction tasks, this one assumes that the acquired knowledge structure will perform well on unseen instances, even at the expense of imperfect performance on the training set. Like most of the induction problems we have examined in previous chapters, the above task allows for disjunctive or noncontiguous concepts. However, the aim is to move beyond the induction of concept descriptions to the generation of an organization of such concepts in memory.

One can view this task as involving search through the space of concept hierarchies. For example, one might cast the induction of a univariate decision tree as search through an AND/OR space, with the AND branches corresponding to attributes and the OR branches corresponding to values. This space is quite large: given a attributes with v values each, it contains some

$$\prod_{i=0}^{a-1}(a-i+1)^{v^i}$$

possible decision trees, which is much larger than the space of possible concepts. This does not take into account the labels on terminal nodes, but the data determine these for any given tree structure. Even so, for a domain with only five attributes having two values each, we have $6^1 \cdot 5^2 \cdot 4^4 \cdot 3^8 \cdot 2^{16} > 1.65 \cdot 10^{13}$ possible trees.

Clearly, one cannot search this space exhaustively when there are more than a few attributes. For this reason, most work on decision-tree induction employs a greedy method that directs search with an evaluation function. Note that, given a consistent set of training data, there exist many alternative decision trees that completely summarize those data. However, one would also like a tree that makes useful predictions about novel instances, and both the search scheme and the evaluation function can influence the predictive ability of the resulting trees. Similar observations hold for other types of conceptual hierarchies.

Given that hierarchy formation involves search, one must organize the search in some fashion, and there are two obvious approaches. One can construct the hierarchy from the root node downward, in a *divisive* manner, or one can build it from the terminal nodes upward, in an *agglomerative* style. Although these may appear similar to the general-to-specific and the specific-to-general methods we examined in Chapter 2, they are actually quite distinct. Divisive and agglomerative methods differ in the order in which they construct their hierarchies, not in the ways they search for concept descriptions embedded in those hierarchies.

Table 7-1. The DCH algorithm: Nonincremental divisive formation of concept hierarchies.

```
Inputs: The current node N of the concept hierarchy.
        A set of training instances ISET.
Output: A concept hierarchy.
Top-level call: DCH(root, ISET).

Procedure DCH(N, ISET)

If the training set ISET is not empty,
Then generate a set of partitions for ISET.
     For each partition P,
         Evaluate P using some metric.
     Select the best partition Best.
     For each cluster JSET in Best,
         Form a node C with JSET as its members.
         Form an intensional description DESC for JSET.
         Associate the description DESC with node C.
         Let KSET be the instances in ISET matched by DESC.
         Make node C a child of node N.
         DCH(C, KSET).
```

One can use any form of concept representation in such a memory organization, and one can invoke any of the induction algorithms from earlier chapters, regardless of the order in which one constructs the hierarchy.

Another central issue in hierarchy formation involves the division of instances into subsets or *clusters*. We will encounter a variety of methods for handling this problem, but we will also find that these techniques are relatively independent of the direction of hierarchy construction and the induction algorithms used to characterize the resulting clusters.

7.2 Nonincremental divisive formation of hierarchies

The first set of methods we will examine for inducing concept hierarchies vary along a number of dimensions, but they have important features in common as well. In addition to the shared representational and performance assumptions already mentioned, all of these methods are nonincremental in nature, and they are divisive in that they construct hierarchies from the top down.

Table 7-1 presents pseudocode for DCH, a basic algorithm for the divisive formation of concept hierarchies that requires all training instances to be available at the outset. The method begins by creating the top node and grows a hierarchy downward from there. At each level, DCH divides the current training instances into alternative disjoint sets or partitions. It then uses an evaluation function to measure the desirability of each partition, and selects the one with the best score.

The algorithm creates one successor node for each cluster of instances in the selected partition, connecting this node to its parent by an IS-A link. DCH can also invoke an induction algorithm to generate an intensional description for each cluster, which it associates with the corresponding node. The method then calls on itself recursively – passing on the new node, the subset of instances in the cluster, and possibly those attributes that have not been used higher in the hierarchy. DCH does this for each successor node, constructing subtrees that summarize each subset of instances in a given cluster. This process continues ever deeper until the algorithm reaches some termination criterion, at which point it pops back to the previous level to handle other clusters in the partition.

7.2.1 Induction of decision trees

As we have mentioned, univariate decision trees constitute an important special case of concept hierarchies. Each node in such a structure specifies a single predictive feature, which it uses as a logical test to direct the sorting of instances. Typically, only terminal nodes specify any predictable features, usually the name of the most likely class. Note that most presentations of decision trees place the values of the discriminating attribute on the branches leading into nodes, and the attribute itself on the node from which they emanate. Our formulation is equivalent but clarifies connections to other work on hierarchy formation.

This organizational scheme supports a straightforward algorithm for the divisive induction of decision trees (DDT) that instantiates the abstract method from Table 7-1. As in the more general algorithm, the initial action is to create a root node for the decision tree (see Table 7-2). The method then generates one partition for each attribute, which for now we assume to be nominal, basing the clusters in each partition on those instances having the same value for that attribute. DDT then evaluates each partition in terms of its ability to discriminate among

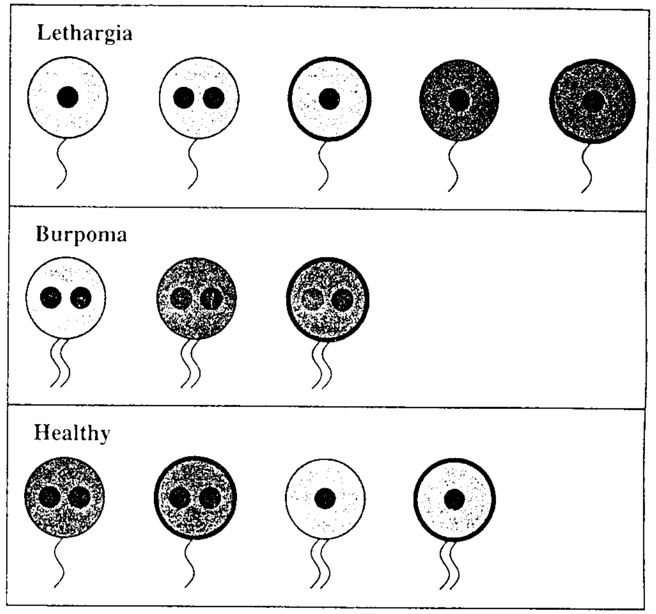


Figure 7-1. Training instances from a cell domain involving four attributes and three distinct classes.

the various classes, generating a class-by-partition contingency table for each alternative. Researchers have used many different evaluation metrics to this end, but all have the similar effect of preferring attributes with values correlated with the class.

The algorithm selects the best such partition, creates a child of the root for each cluster in the partition, and associates the appropriate attribute-value pair as its predictive summary description. If all instances in a cluster belong to the same class, then DDT stores the most likely class as the predictable feature for its associated node. Otherwise, the algorithm calls on itself recursively, passing on the instances in the cluster to use in building a subtree with the current node as its root.

Let us examine this algorithm's behavior on a variation of the cell domain from previous chapters. Figure 7-1 presents a set of training instances for three classes of patients – those who have the disease `lethargia`, those with `burpoma`, and those who are `healthy`. Each instance describes a cell from the patient in terms of four attributes –

Table 7-2. The DDT algorithm: Divisive induction of univariate decision trees.

```
Inputs: The current node N of the decision tree.
        A set of classified training instances ISET.
        A set of attributes and their values ASET.
Output: A univariate decision tree.
Top-level call: DDT(root, ASET, ISET).

Procedure DDT(N, ASET, ISET)

If the training set ISET is empty,
Then label terminal node N as UNKNOWN.
Else if all instances ISET are in the same class or
    If the attribute set ASET is empty,
    Then label terminal node N with the class name.
    Else for each attribute A in the attribute list ASET,
             Evaluate A according to its ability to
                discriminate the classes in ISET.
         Select attribute B with the best evaluation score.
         For each value V of the best attribute B,
             Create a new child C of node N.
             Place the attribute-value pair (B, V) on C.
             Let JSET be the ISET instances with value V on B.
             Let KSET be the ASET attributes with B removed.
             DDT(C, KSET, JSET).
```

the number of `nuclei`, the number of `tails`, the `color`, and the cell `wall`. Since each attribute takes on two possible values, there exist 16 possible instances, only 12 of which occur in the training set.

Figure 7-2 shows two versions of the decision tree that DDT generates from these training instances, with (a) depicting the standard notation with attributes on parent nodes and values on links, and (b) graphically showing the type of instances matched by each node. The algorithm's first step in constructing this tree involves examining the data and comparing the four attributes in terms of their ability to discriminate the classes. This requires some evaluation function; here we will assume the metric

$$\frac{1}{n}\sum_v C_v \quad ,$$

where n is the number of training cases sorted to the current node, the summation occurs over all values of the candidate attribute, and C_v is the number of instances with value v that are correctly classified by the most frequent class for that value. This metric has a maximum score of 1 (for a perfectly discriminating attribute) and a minimum of 0 (for one with no discriminating ability).

For the data in Figure 7-1, the number of `tails` receives the best score, in this case $\frac{5+3}{12} = 0.667$. As a result, DDT partitions the instances along this dimension and creates two children of the root node, each with a different value of this attribute. The algorithm sorts all instances of the `lethargia` class down the `one tail` branch, along with some cases of the `healthy` class. Similarly, it sends all instances of `burpoma` down the `two tail` branch, along with the remaining `healthy` patients.

The algorithm calls on itself recursively to handle each of these instance clusters. Of the remaining three attributes, it finds the cell `color` to be most discriminating for the first set, with the score $\frac{3+2}{7} = 0.714$. Thus, it creates one node for each value of this attribute, storing these values as their predictive features. All instances in the cluster with `light color` are members of the `lethargia` class; thus, DDT does not extend the decision tree further than the node for this set. In contrast, instances in the cluster with `dark color` contain some patients with `lethargia` and some `healthy` ones, so another split is needed. In this case, the most useful attribute is the number of `nuclei`, with the score $\frac{2+2}{4} = 1$. Thus, DDT creates two children that test on this feature, which also lead to pure nodes.

At this point, the algorithm returns to the second cluster of instances generated by its initial decision, which share the attribute-value pair `two tails`. Since this set contains members of two classes, DDT selects another attribute to distinguish between them. In this case it also decides on the number of `nuclei`, although this occurs at an entirely different level of the tree than the other use of the attribute. Upon creating nodes for each value of this attribute, the algorithm finds that all instances in the cluster with `one nucleus` are `healthy`, whereas all those with `two nuclei` have `burpoma`. Thus, there is no need to extend the tree further downward, and the algorithm halts. The figure shows the order in which DDT generates each node, along with the class and number of training instances associated with each terminal node.

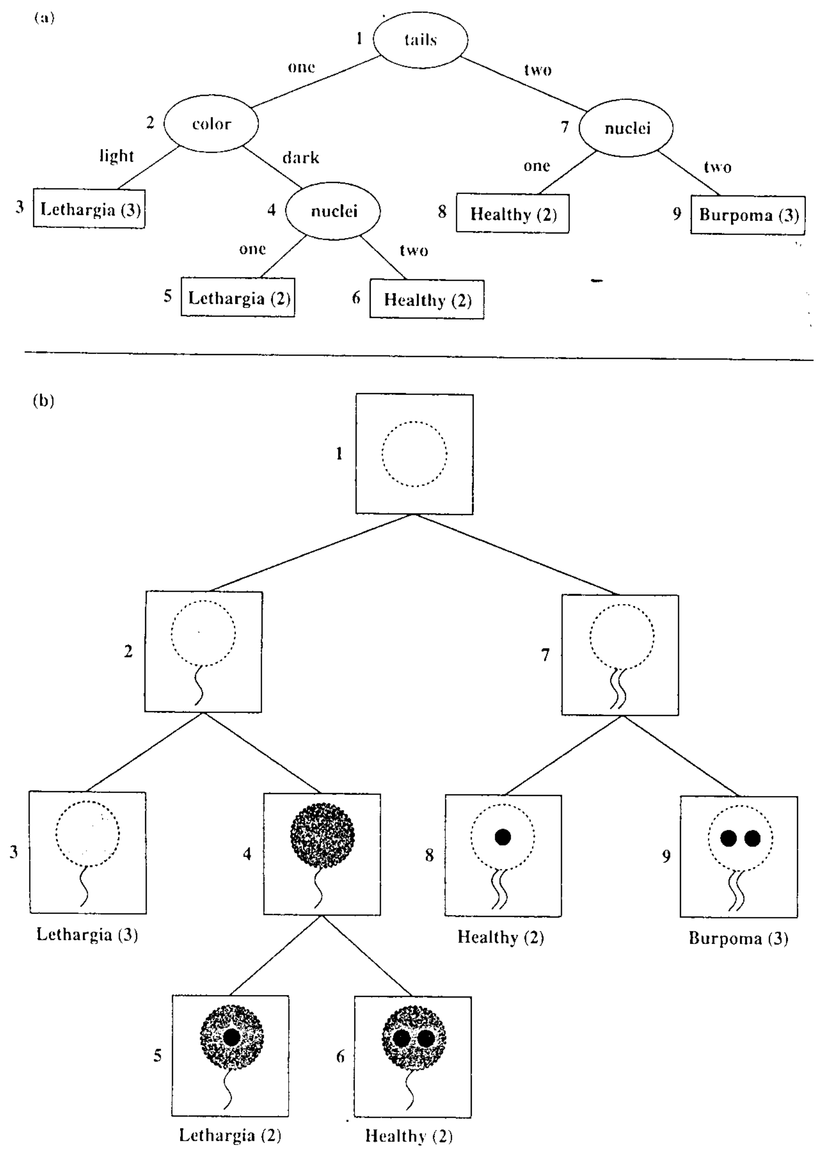


Figure 7-2. A decision tree generated by the DDT algorithm for the training instances from Figure 7-1. Each terminal node specifies an associated class name and the number of training cases covered. Numbers next to the nodes indicate the order in which the algorithm generated them. The notation in (a), which associates attributes with nodes and values with links, is equivalent to (b), which shows the instances that each node covers.

7.2.2 Variations on decision-tree induction

Upon selecting a nominal attribute, the DDT algorithm creates one branch for each value of that attribute. However, when attributes take on many possible values, one can imagine domains in which trees that incorporate *sets* of values would give simpler output and greater classification accuracy. Naturally, the basic technique can be extended to construct trees of this sort. The simplest scheme involves creating a binary decision tree in which two branches emerge from each internal node, one specifying a particular value for an attribute and the other specifying its negation.

Modifying the DDT algorithm to construct such a binary tree is not difficult, but it does introduce additional search. The cost of the basic method is linear in the number of attributes, since at each decision point, it must iterate through all unused attributes. In order to construct a binary tree of this sort, DDT must also iterate through each *value* of each attribute, giving a cost that is linear in both the number of attributes and the number of values. In addition, if an attribute has more than two values, it must reconsider splitting on its remaining values when recursing down the negated branch. We cannot illustrate this method with the cell domain, since its attributes take on only two values each.

One can extend this scheme further by constructing binary trees in which each branch contains an arbitrary subset of an attribute's values. Again, there is no problem in modifying the DDT algorithm to induce such trees, and some variants take precisely this approach. However, this extension is even more expensive, since it must consider all two-set partitions of values for an attribute. An even more extreme variant allows decision trees with an arbitrary number of branches, in which each branch specifies an arbitrary set of an attribute's values; this method introduces additional costs into the induction process but may produce simpler or more accurate hierarchies.

Although decision trees were originally designed to handle nominal (symbolic) data, they can easily be extended to deal with integer-valued or real-valued attributes. Given N instances, one first sorts those instances in terms of increasing values on the numeric attribute A. Each successive pair of values V_k and V_{k+1} defines a possible threshold test, $A > (V_k + V_{k+1})/2$, that partitions the instances into two sets. Thus, one computes an evaluation score for each of the $N - 1$ tests, retaining the best such test and its score. Finally, one compares this test's score to those obtained for other attributes, whether nominal or numeric, and selects the best-scoring test for the current level in the decision tree.

One can view this approach to building decision trees as partitioning an N-dimensional space, with each terminal node representing one hyper-rectangle and each test specifying one of its faces.[2] The technique noticeably increases the complexity of the decision-tree building process, but not outrageously so. Given M numeric attributes and I instances, the cost at each level of the tree is $M(I-1)$ times an equivalent task involving the same number of nominal attributes. The cost of constructing the overall tree is more difficult to compute, since a numeric attribute must be reconsidered for additional tests lower in the tree, unlike nominal ones. Many of the tests reported in the literature on natural domains have relied on this extension.

One can even adapt the basic DDT algorithm to construct decision trees from unsupervised training data. In this scheme, the modified algorithm still generates a set of alternative partitions based on the various attributes. However, it cannot evaluate them on their ability to discriminate among the classes, since there is no class information. Instead, the method measures a partition's ability to discriminate among the values of those attributes not used to form the partition.

Thus, given a domain with A attributes, one of which has been used to create partition P, the unsupervised algorithm treats each of the remaining $A-1$ attributes as 'class names' in turn, measuring the partition P's ability to discriminate among the values of the first attribute, then the second, and so forth. The method then selects the partition with the best *average* discriminating ability across all the attributes, creates nodes for each cluster in this partition, and recurses as in the standard algorithm.

This approach to evaluating partitions has clear links to the natural performance task for unsupervised induction, which involves predicting any attribute that is missing from a test instance. The univariate nature of decision trees introduces difficulties for instances that omit attributes used as tests but, as we discuss in Chapter 8, there are a number of remedies for this problem.

2. Thus, decision-tree methods form decision boundaries that are parallel to the axes of the instance space, like the conjunctive methods in Chapter 2.

7.2.3 Formation of multivariate concept hierarchies

Other variations on the DCH framework construct hierarchies that are *multivariate* in nature, in that each node or concept incorporates multiple predictive features. One such scheme still generates partitions on the basis of a single attribute, but does not necessarily include it as one of the predictors stored on children. Instead, it invokes one of the algorithms for inducing logical descriptions that we described in Chapter 2, such as HGS or HSG. These routines generate a conjunctive summary for each cluster of instances, using members of other clusters as negative training cases. The resulting descriptions are likely to include the attribute used to formulate the partition, but this is not guaranteed. They are also likely to be more specific, and thus more conservative in making predictions, than the single-attribute approach. For example, a new instance might not match the description associated with any child, forcing the performance module to reply 'unknown' or to predict the most likely class at the current level.[3]

Actually, one can combine this partitioning scheme with any conceptual representation and with any algorithm for inducing simple concept descriptions. After generating a partition, the DCH algorithm could instead use the LMS or IWP technique (Chapter 3) to generate a threshold concept for each cluster, using members of that cluster as positive instances and members of others as negatives. Another alternative is to use an instance-averaging method or a naive Bayesian classifier (Chapter 4) to generate one competitive concept description for each cluster of training cases.

Recall that, when used in isolation, techniques like HSG, LMS, IWP, Bayesian classifiers, and simple instance averaging are severely limited in their representational ability, and thus in their ability to learn. However, the current approach should not suffer from such limitations because the hierarchy stores knowledge at multiple levels of abstraction. Subdivisions at lower levels overcome the representational drawbacks at a given level, letting the hierarchy as a whole represent complex extensional definitions even though each level only describes simple ones. For the same reason, there seems no reason to invoke more powerful

3. One can even delay evaluation of partitions until after finding summary descriptions for each set of clusters, then use evaluation criteria that measure characteristics of these descriptions, like 'simplicity', to select among the partitions. This idea is common in techniques for *conceptual clustering*.

Table 7-3. The DMT algorithm: Divisive induction of multivariate trees.

```
Inputs: The current node N of the concept hierarchy.
        A set of classified training instances ISET.
Output: A multivariate concept hierarchy.
Top-level call: DMT(root, ISET).

Procedure DMT(N, ISET)

If all instances ISET are in the same class,
Then label terminal node N with the class name.
Else for each class C occurring in ISET,
          Let JSET be the members of ISET having class C.
          Form a node S with JSET as its members.
          Form an intensional description DESC for JSET.
     For each cluster JSET and associated S and DESC,
          Let KSET be the instances in ISET matched by DESC.
          Form a revised description DESC based on KSET.
          Associate the description DESC with node S.
          Make the node S a child of node N.
          DMT(S, KSET).
```

induction methods in this context, since they would add little to the overall effectiveness.

Another approach to forming multivariate concept hierarchies, less common in the literature, uses an even simpler partitioning scheme. Table 7-3 presents pseudocode for DMT (divisive induction of multivariate trees), another specialization of the DCH algorithm. At each level, this method simply groups instances by class names, then uses one of the above induction methods to generate intensional descriptions for each cluster. Next, DMT uses these descriptions to reassign instances, which leads to a revised partition and new summaries. In most situations, the second set of descriptions will not completely discriminate among the classes, so the revised partition will differ from the original one, as will the two sets of descriptions.

As usual, if a cluster contains instances from more than one class, DMT calls on itself recursively to further subdivide the data and to form more specific concept descriptions for each subclass. This approach makes less sense if one's goal is a univariate hierarchy, since the

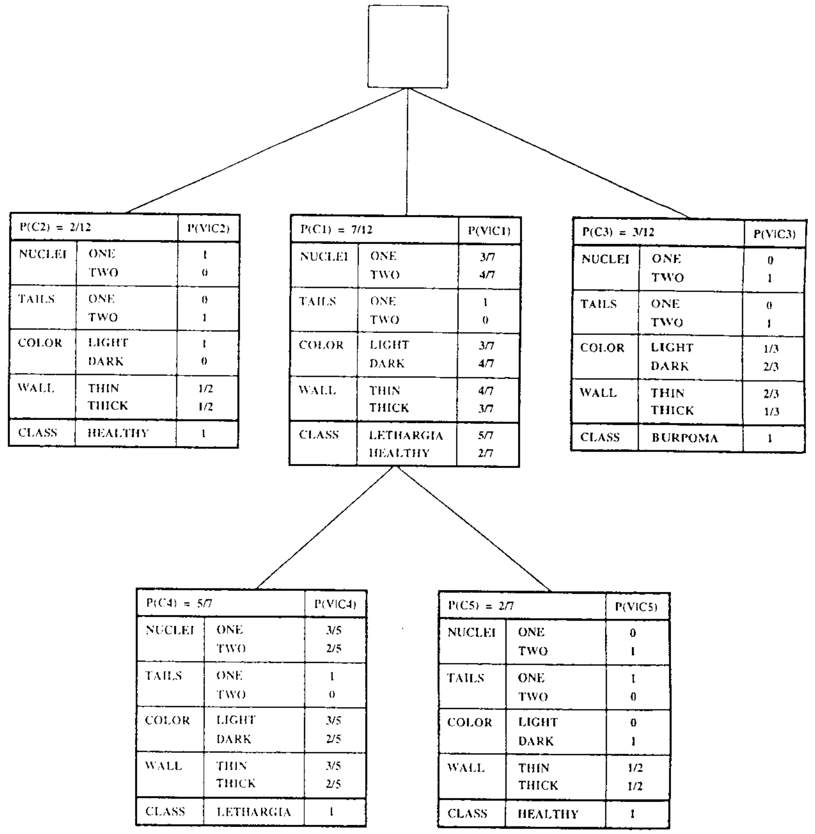


Figure 7-3. A multivariate concept hierarchy generated by the DMT algorithm for the training instances from Figure 7-1. Each node contains a probabilistic summary, which a Bayesian classifier uses to sort instances and to make predictions.

DDT algorithm generates one more directly. Nor can one easily adapt this scheme to handle unsupervised training data, since it relies on class information to form partitions. On the other hand, it provides an interesting approach to generating multivariate hierarchies from supervised training data.

Figure 7-3 depicts the concept hierarchy generated by DMT for the data in Figure 7-1 when it uses a Bayesian classifier as the induction subroutine. As in most probabilistic concepts, each node includes a base rate and a conditional probability for each attribute-value pair. In generating this tree, the algorithm first partitions the training instances into three clusters, one for each class, and then uses a Bayesian classifier to produce a probabilistic summary for each set. The resulting descriptions predict the correct class names for all instances of `lethargia` and `burpoma`, as well as for the second two `healthy` cases. However, it assigns the first two `healthy` cells to the `lethargia` class.

In response, DMT generates a revised partition based on these three groups of instances, then computes a revised probabilistic summary for each one. Because one of the clusters contains instances from two classes, it calls itself recursively on this subset of instances. Thus, the algorithm creates one partition for the five `lethargia` cases and another for the two `healthy` instances, after which it invokes the Bayesian classifier a second time to produce descriptions for each such cluster. This time the probabilistic summaries correctly predict the class associated with each instance, so DMT halts its construction of the probabilistic concept hierarchy.

7.2.4 Iterative optimization

A third approach to clustering, more common for unsupervised tasks, is known as *iterative optimization*. The basic method requires the user to specify the number of clusters at each level, although some variants iterate through different numbers on their own. The basic algorithm begins by selecting k instances to serve as seeds for each cluster. Typically the seeds are selected randomly, but more directed strategies are also possible. Next the method selects an unassigned training case and tentatively adds it to each cluster; after comparing the alternatives on some evaluation metric (e.g., distance to the cluster mean), it selects the best cluster and assigns the training case to it. The algorithm repeats this process for each nonseed instance in the current set.

At this point, the routine invokes some induction method to generate a summary description for each resulting cluster of instances. In some versions, it constructs this incrementally after each incorporation and uses the descriptions in its assignment decisions. In either case, iterative optimization then uses the summary description for each cluster to select or construct a new seed for that cluster. The algorithm then repeats the assignment process, placing each nonseed instance in one of the clusters, possibly a different one than on the previous pass. This iterative procedure continues until it reaches some halting condition, typically when it selects the same seeds twice in a row, which suggests that the partition and summaries have stabilized. Once iterative optimization has settled on a partition, one can recursively apply the method to produce lower levels of the hierarchy if desired.

As we have noted, iterative optimization is most widely used for unsupervised learning tasks, when the simpler methods do not apply. The scheme is most closely associated with nearest neighbor or instance-based methods, where the summary description for each cluster is either the set of training instances in that cluster or their average; this latter approach is often called *k-means clustering*. However, one can also combine the above technique with probabilistic induction methods, threshold learning algorithms, and even logical induction schemes. An important special case, known as the *expectation maximization* (EM) algorithm, probabilistically assigns instances to each cluster, updating the probabilistic summaries by the fractional amount assigned to each.

Unlike the simpler clustering methods we have discussed, iterative optimization involves a hill-climbing search through the space of possible partitions. A number of factors can affect the details of this search, including the initial seeds, the halting criterion, and even the order in which one assigns instances to clusters. But despite its computational cost and its sensitivity to these factors, the method has seen widespread use within algorithms for unsupervised learning.

7.3 Incremental formation of concept hierarchies

Now that we have reviewed some nonincremental algorithms, we can consider some incremental approaches to forming concept hierarchies. As we have noted earlier, incremental learning leads naturally to the integration of learning with performance. In most incremental systems, action by the performance component drives the learning element, and this also holds for the organization of memory. In Section 7.1, we noted that performance for a hierarchy-forming system involves sorting instances down through the concept hierarchy. As a result, it seems natural to acquire such hierarchies in a top-down fashion as well, and incremental methods typically construct their hierarchies in a divisive manner. One can view methods of this sort as carrying out an incremental hill-climbing search through a space of concept hierarchies.

Table 7-4. The ICH algorithm: Incremental formation of concept hierarchies.

```
Inputs: A set of training instances ISET.
Output: A concept hierarchy.

Procedure ICH(ISET)

Create a root node N based on the first instance J in ISET.
For each instance I other than J in ISET,
    ICH-Aux(N, I).
Return the concept hierarchy with the root node N.

Procedure ICH-Aux(N, I)

Let FLAG be False.
For each child C of N,
    If I matches C's predictive features well enough,
    Then let FLAG be True.
         If I mismatches C's predictable features,
         Then [expand the concept hierarchy downward]
              Create a child P of node N.
              Base the description of P on C and I.
              Create a child D of P based on I.
              Make C a child of P rather than N.
         Else modify the description of C based on I.
              ICH-Aux(C, I).
If FLAG is False,
Then [expand the concept hierarchy outward]
     Create a child E of node N based on instance I.
```

Because such methods can be sensitive to the training order, many of them include bidirectional learning operators that can reverse the effects of previous learning should new instances suggest the need.

Table 7-4 presents pseudocode for ICH, an abstract algorithm for the incremental divisive formation of concept hierarchies. The top-level algorithm uses the first training case to initialize the hierarchy (attaching initial predictive and predictable descriptions) and then iterates through the remaining instances, calling on ICH-Aux to carry out the main work. Given an existing hierarchy and an instance I, the subroutine sorts I through memory, starting at the root node. At a given node N, ICH-Aux compares I to the predictive features stored at each child C of N.

If the instance does not match any of these well enough, this suggests the need for a new category; thus, the algorithm creates a new child of N based on the instance, making the hierarchy wider at this level.

In contrast, if the instance I matches the predictable features for child C sufficiently well, the algorithm incorporates I into that node's description and recurses. If the predictive features match but the predictable ones do not, this suggests the need for making further distinctions. Accordingly, ICH-Aux extends the hierarchy downward by creating a new node P that becomes a child of N and the parent of C and a second new node, D, based on I. A special case occurs when the algorithm reaches a terminal node with predictable features that differ from the training case. The ICH algorithm calls on ICH-Aux anew for each training case, expanding the tree in breadth and depth as experience suggests.

7.3.1 Incremental induction of decision trees

Let us examine how one can adapt the above approach to the incremental induction of univariate decision trees. The specialized algorithm, which we will call IDT (incremental induction of decision trees), initializes the tree to contain a single root node with no predictive features and a single predictable feature, the class of the instance. It also stores the description of the instance with the node for use during learning. If the second instance has the same class as the first, then IDT takes no action, but if they disagree, it compares the new instance with the stored one to find their differences. The algorithm selects one difference at random and uses the attribute-value pairs as predictive tests on the two nodes it creates in expanding the tree downward. IDT stores the first instance with one child and the second instance with the other.

The algorithm repeats this process with successive training cases, sorting them through memory until they fail to match any of the nodes at a given level or until they reach a terminal node. In the former situation, IDT creates a new child with a predictive test that incorporates the same attribute as other nodes at that level; it also stores the training instance with this new node. In the latter case, the algorithm compares the class predicted by the terminal node to the class of the instance. If they disagree, IDT extends the tree downward, using some difference between the new instance and the stored one to identify a predictive attribute for the new nodes. If they agree, the algorithm makes no changes to memory.

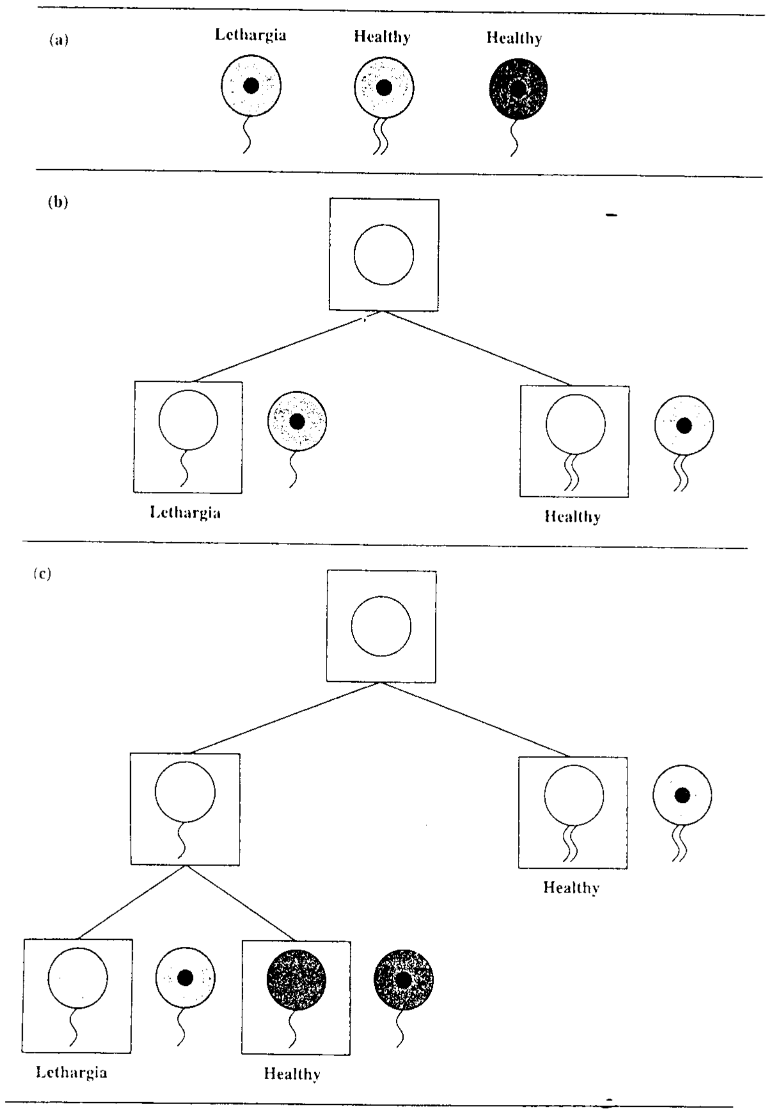


Figure 7-4. (a) Three training instances in the order presented to the IDT algorithm; (b) the decision tree generated by IDT after the first two training cases; and (c) the decision tree produced after processing the third case. Each node includes an associated instance for use in learning, shown immediately to its right.

Figure 7-4 shows a decision tree at two stages during its construction by the IDT algorithm when presented with training instances from the cell domain. The first training instance, a member of the `lethargia` class, leads to an initial one-node tree (not shown). Upon encountering the second training case, from a `healthy` patient, the algorithm finds that the predicted and observed class disagree. As a result, it compares the new and the stored instance, which reveals only one difference: the new instance has `one tail`, whereas the earlier one had `two tails`. Thus, IDT creates two new nodes, one with `one tail` as its predictive feature and `lethargia` as its predictable class, the other with `two tails` as its predictive feature and `healthy` as its associated class. Figure 7-4 (b) shows this state of the decision tree.

Upon processing a third instance, this one also from the `healthy` class, IDT finds that the case matches the predictive test for the leftmost child, causing it to sort the instance down this branch. However, the instance's class again fails to agree with the predicted one, causing a second discrimination to occur. This time the only difference between the stored and new instance is the `color`, leading to two new children with predictive tests based on this attribute. One of these predicts the `lethargia` class, whereas the other predicts `healthy` cells. This process continues, with the method extending the decision tree downward only when an instance's class differs from that stored on the node that it reaches.

Of course, we have simplified this example for the sake of clarity. In a domain with attributes having more than two possible values, the algorithm would also broaden the tree when an instance fails to match the descriptions of any children. More important, the example uses a training order in which IDT finds only one difference between instances. Given a benign training order, the algorithm will create the same decision tree as that generated by its nonincremental counterpart DDT. However, some training orders generate multiple differences, and the algorithm might well select one that it would later regret, since it has no mechanism for retracting such choices.

One can easily extend this general approach to learn discrimination networks from unsupervised training data. Rather than storing the class name as the predictable feature for a terminal node, the method instead stores one or more features of the instance itself. These may or may not include features used as predictors along the path to that node.

The modified algorithm bifurcates the tree and extends it downward whenever features of a training instance I differ from the predictable features stored at the terminal node to which I has been sorted.

7.3.2 Heuristic revision of univariate decision trees

The above schemes, whether applied to supervised or unsupervised data, will construct a decision tree or discrimination network that summarizes the instances. However, the random selection of predictive features means that the resulting hierarchies may be more complex than necessary. In addition, one can easily mislead IDT by adding noise to the training data. An obvious response to both problems is to use a more conservative method that collects statistics on the various features across a number of instances, as does the IHC method in Chapter 2.

For example, one might store with each node N the number of instances of each class that have passed through N, along with the number of times it has observed each attribute-value pair in instances sorted through N. Recall that the nonincremental DDT generates contingency tables directly from a set of instances; this scheme stores similar tables in the tree itself, opening the way for incremental updates.

Table 7-5 presents the algorithm HDT (heuristic revision of decision trees), which updates the class and attribute-value counts each time it sorts an instance through a given node. Occasionally, the algorithm sorts an instance having a value V on attribute A to a node that has never encountered this value. In this case, it creates a new branch labeled V, pointing to a new terminal node that predicts the observed class.

In addition to updating the counts at each node, HDT also recomputes the evaluation score for each predictive attribute after incorporating a new instance. For terminal nodes, this can lead the algorithm to extend the decision tree downward, adding new branches based on the values of an attribute that now appears useful. For nonterminal nodes, it can lead the system to replace an existing attribute with another one that now appears more discriminating. In this case, HDT removes the existing subtree below this node and gradually constructs a new subtree from additional training cases that incorporates the new attribute. In taking such a drastic step, the method loses information about previous instances. This effectively means trading smaller processing costs per instance for more instances during the tree-construction process.

Table 7-5. The HDT algorithm: Heuristic incremental induction of univariate decision trees.

```
Inputs: A set of classified training instances ISET.
Output: A univariate decision tree.

Procedure HDT(ISET)
Create a root node N for the decision tree.
For each training instance I in ISET,
    Let C be the class of instance I.
    HDT-Aux(N, I, C).
Return the decision tree with root node N.

Procedure HDT-Aux(N, I, C)
Use I to update the contingency table for node N.
For each attribute A in the attribute list ASET,
    Compute an evaluation score $E_A$ for attribute A.
Select the attribute B with the best score $E_B$.
If B differs from the current attribute Y on N,
Then replace Y with B as the attribute associated with node N.
     Remove all existing subtrees of node N.
     If the best score $E_B$ is sufficiently high,
     Then for each value V of B stored on N,
              Create a child S of node N.
              Place the attribute-value pair (B, V) on S.
              Initialize the counts on S to zero.
Else let A be the attribute associated with N.
     Let V be the value of instance I on attribute A.
     If N has a successor S with test (A, V),
     Then HDT-Aux(S, I, C).
     Else create a new successor S of N.
          Place the attribute-value pair (A, V) on S.
          Initialize the contingency table for S to contain
             zero entries for each attribute-value pair.
```

A more sophisticated variant of this algorithm, which we will call HDT′, stores all training instances at the terminal nodes to which they were sorted, thus avoiding loss of information. Upon deciding to replace attribute A at a node with an attribute B that occurs lower in the hierarchy, the modified method moves the B test to the current node, moves A to the level immediately below, and recursively pushes attributes lower in the hierarchy one level down. If B does not occur in any subtree of the current node, HDT′ simply introduces it and pushes all subtrees downward by one level. This procedure is guaranteed to produce the same decision tree as DDT on any given training set, but only at the cost of reprocessing more instances than HDT.

Both approaches incorporate the notion of bidirectional operators that are central to robust methods for incremental hill climbing. Even using statistical techniques, nonrepresentative training sequences can mislead a learning algorithm; thus, it should be able to change its mind, and the operator for replacing one predictive attribute with another gives this capability.

7.3.3 Incremental formation of multivariate hierarchies

The methods we have examined let one construct univariate decision trees and discrimination networks in an incremental manner, but they do not support the formation of multivariate hierarchies. When generating structures that incorporate multiple features at each level, it makes little sense to use the values of a single attribute to partition the training instances. The central issue in such methods concerns when to create a new sibling at a given level.

Given classified training data, one can use errors in the class prediction to drive the creation of new nodes, as in many methods we have examined for incremental learning. Thus, one can employ incremental hill-climbing versions of the DMT algorithm, which partitions instances by their class name and constructs subtrees only when the descriptions induced for the current level fail to completely discriminate among the classes. One can instantiate this scheme using any of the incremental methods for inducing logical, threshold, or competitive concepts from Chapters 2 through 4.

Another approach, not driven by errors, draws on the incremental ICD algorithm from Chapter 5, which creates new disjuncts when a training instance differs enough from existing descriptions. Table 7-6 summarizes an algorithm that employs this idea to constrain the incremental formation of multivariate hierarchies (IMT). This method sorts a new training instance downward through its hierarchy, at each level deciding whether the instance is 'close enough' to the predictive de-

Table 7-6. The IMT algorithm: An incremental method for the formation of multivariate concept hierarchies.

```
Inputs: A set of training instances ISET.
Output: A concept hierarchy.

Procedure IMT(ISET)
Create a root node N based on the first instance J in ISET.
For each instance I other than J in ISET,
   IMT-Aux(N, I).
Return the concept hierarchy with the root node N.

Procedure IMT-Aux(N, I)
If N is a terminal node,
Then replace N with a node S that summarizes N and I.
     Create a node M based on instance I.
     Store N and M as children of S.
Else modify N to incorporate instance I.
     For each child C of node N,
         Compute the desirability of placing I in C.
     Let B be the child with the best score $S_B$.
     Let $S_{new}$ be the score for creating a new node.
     Let $S_{merge}$ be the score for merging B with its sibling D.
     Let $S_{split}$ be the score for splitting B into its children.
     If $S_{new}$ is the best score,
     Then create node R with a description based on I.
          Store node R as a child of node N.
     Else if $S_B$ is the best score,
          Then IMT-Aux(B, I) (place I in category B).
          Else if $S_{merge}$ is the best score,
               Then Merge(B, D, N).
                    IMT-Aux(B, I).
               Else Split(B, N).
                    IMT-Aux(N, I).
```

scription of one of the existing children. If not, IMT creates a new child at the current level, basing its description on that of the instance. In contrast, if the instance matches one of the existing children well enough, the algorithm incorporates the instance into that child's description (by changing logical constraints, weights on threshold units, or altering competitive summaries) and recurses to the next level.

Although these steps are sufficient to form a multivariate hierarchy in an incremental manner, by themselves they would be quite sensitive to the instance presentation, creating quite different hierarchies from different orders of the same data. For example, if the first instances are all dogs, the scheme would create subcategories of these at the top level. When it finally encountered instances of cats, it would create one category for them at the top level. However, it would still have all the dog instances at this same level, when one would prefer them grouped under a separate category.

IMT includes two additional operators to help it recover from such nonoptimal hierarchies. At each level of the classification process, the system considers *merging* the node that best classifies the new instance with its various siblings. If the resulting partition is better than the original according to some evaluation function, IMT combines the two nodes into a single category, although still retaining the original nodes as its children. This transforms a level with N nodes into one having $N - 1$ nodes.

The algorithm also incorporates the inverse operation of *splitting* nodes. At each level, if IMT decides to classify an instance as a member of an existing category, it also considers removing this category and elevating its children. If this action leads to an improved partition, the system changes the structure of its hierarchy accordingly. Thus, if one of N nodes at a given level has M children, splitting this node would give $N + M - 1$ nodes at this level. Taken together, these two operators can help IMT recover from early decisions that would otherwise produce undesirable hierarchies, in the same way that alternative bidirectional operators aid other incremental hill-climbing methods.

7.4 Agglomerative formation of concept hierarchies

A quite different approach to the construction of concept hierarchies operates in an *agglomerative* manner, grouping instances into successively larger clusters rather than refining them, as in the divisive scheme. Although one can run such methods on supervised training data, they are typically used on unsupervised learning tasks. Also, they are nearly always nonincremental in nature, requiring that all instances be present at the outset.

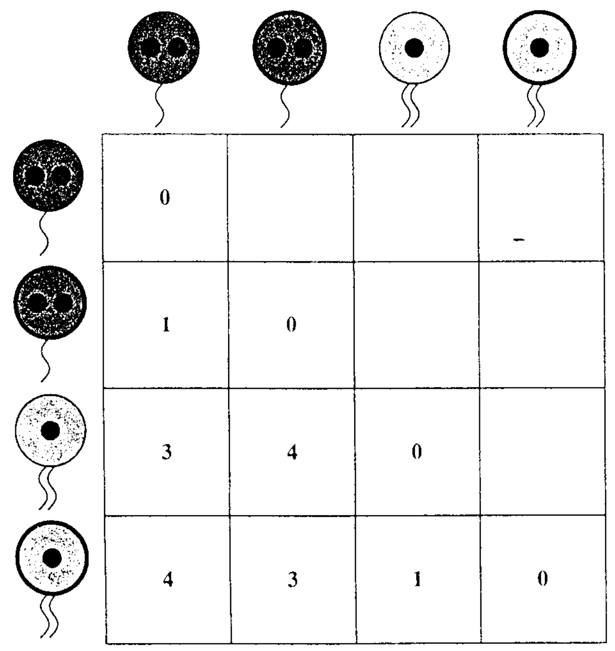
Table 7-7. The ACH algorithm: A nonincremental agglomerative method for generating concept hierarchies.

```
Inputs: A set of training instances ISET.
        A set MATRIX that gives the distances between
          each pair of training instances in ISET.
Output: A concept hierarchy.

Procedure ACH(ISET, MATRIX)
Let (A,B,DIST) be the MATRIX entry with the least distance.
Create a new node C with A and B as its children.
Generate a description $D_C$ based on $D_A$ and $D_B$.
Remove A and B from ISET.
If ISET is empty,
Then return the tree with top node C.
Else add the node C to ISET.
     Remove all entries containing A and B from MATRIX.
     For each entry I in ISET,
         Compute the distance DIST between $D_I$ and $D_C$.
         Insert the triple ($D_C$,$D_C$,DIST) in MATRIX.
     ACH(ISET, MATRIX).
```

Researchers in *numerical taxonomy* have developed a variety of such algorithms, but here we will focus on a typical one, which we will call ACH (agglomerative formation of concept hierarchies). Table 7-7 presents pseudocode for this method, which accepts a set of training cases and a matrix that specifies all pairwise distances between the instances. Given this input, ACH finds the closest pair of entries A and B, which may be observed instances or, on later calls, conceptual summaries for sets of instances. The algorithm combines the two entries into a new cluster C, storing A and B as its children in the hierarchy and generating an intensional description for C. For numeric data, the simplest approach for agglomerative algorithms is to use an averaging technique that computes a single prototype for C based on the prototypes for A and B. However, one can also store the instances themselves or generate a logical, threshold, or probabilistic summary.

Next the algorithm checks to see if any entries remain to be incorporated. If not, ACH halts, returning the entire hierarchy it has generated

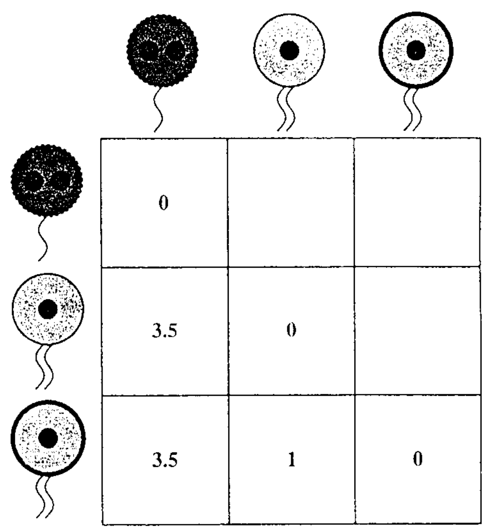


0			–
1	0		
3	4	0	
4	3	1	0

Figure 7-5. The initial distance matrix generated by the ACH algorithm from the four `healthy` instances in Figure 7-1, using the city block distance metric or Hamming distance.

along the way. If entries remain, it removes all pairs containing A and B (since they are now covered by C) and calculates all pairwise distances between C and the remaining entries, sorting them into the ordered list. ACH then calls itself recursively on the new set of pairs, combining the closest pair of entries, adding a new node to the hierarchy, and so forth, until it has combined all entries into a single taxonomic structure.

Although agglomerative methods are typically run on numeric data, we will demonstrate ACH's behavior in a nominal domain for the sake of comparison with other methods. Figure 7-5 shows the distance matrix generated for the four `healthy` cells from Figure 7-1, using a city block measure (i.e., Hamming distance) that contributes 1 for each mismatched feature and 0 for each matched attribute. The matrix shows that two pairs of cells differ on only one feature, but that others have greater distances. Thus, ACH selects one of the nearest pairs at random to form the first cluster, producing the revised matrix in Figure 7-6.



0		
3.5	0	
3.5	1	0

Figure 7-6. The second distance matrix generated by the ACH algorithm, after creation of the first cluster from the four `healthy` instances in Figure 7-1, using a probabilistic version of the city block distance metric.

To compute the distances in this second matrix, ACH must take some stand on the distance between instances and clusters. Here we assume a variant of the city block metric that assumes probabilistic cluster descriptions and gives credit for partial values. For example, the probabilistic average of the two merged instances can be stated as

`tails`	`one`	1	`two`	0
`nuclei`	`one`	0	`two`	1
`color`	`light`	0	`dark`	1
`wall`	`thin`	$\frac{1}{2}$	`thick`	$\frac{1}{2}$

Thus, the distance between this summary and the third cell, which has **`two tails`**, **`one nucleus`**, **`light color`**, and a **`thin wall`**, would be $1+1+1+\frac{1}{2} = 3.5$, which equals the average distance between the two members of the cluster and the instance. This measure also lets the method compute the distance between pairs of cluster summaries.

In any case, the distance between the pair of remaining instances is much less than either's distance to the initial cluster, which leads ACH to combine them into a second cluster with its own probabilistic de-

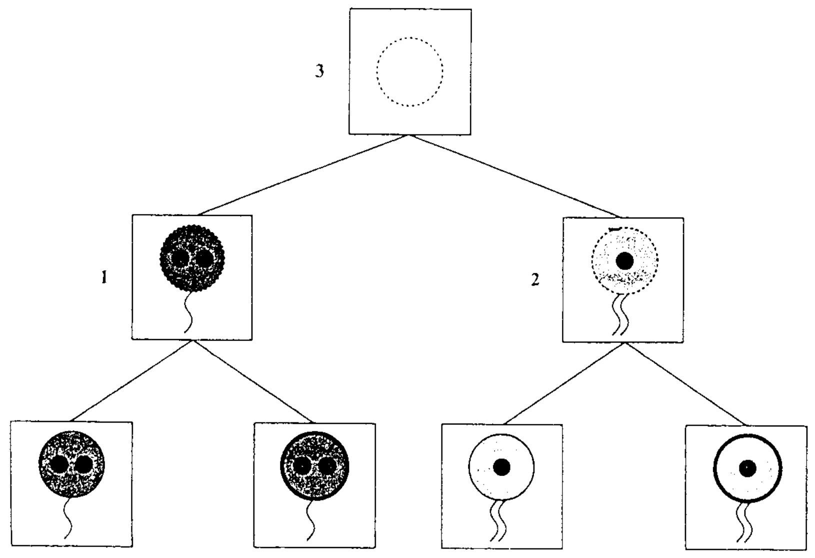


Figure 7-7. A concept hierarchy formed by the ACH algorithm from the four healthy instances in Figure 7-1, using a probabilistic version of the city block distance metric. Numbers next to the nodes indicate the order in which ACH generated them. The probabilistic descriptions for nodes are not shown.

scription. This in turn causes the algorithm to produce a new matrix with distances between the clusters. However, as there exist only two entries, ACH simply attaches them to a root node and returns the resulting hierarchy. Figure 7-7 shows the resulting structure, along with the order in which nodes are created.

Clearly, the ACH algorithm constructs a binary concept hierarchy with exactly two children for each nonterminal node. One can modify the algorithm to find the k nearest entries on each recursion, thus generating a branchier tree. Another alternative is to use some threshold to determine the minimum distance between entries before placing them in the same cluster. If the user is willing to specify the branching factor of the hierarchy at the outset, he can even employ iterative optimization (Section 7.2.4) to determine the clusters that are merged at each level.

There exist many dimensions of variations on the basic agglomerative approach, the most important involving the manner in which they measure distance. The version we saw above found a central tendency

for each cluster and computed distances between them. However, one can describe clusters using the training cases themselves and compute the distance between the two nearest instances in each cluster. For some data sets, this strategy can generate very different hierarchies from those produced with schemes that use central tendencies. The distance measure itself can also greatly affect the resulting taxonomies, and researchers have explored a wide variety of such metrics.

Research in numerical taxonomy, where methods like ACH originated, has typically been aimed at exploratory data analysis. Thus, users have often been more concerned with forming understandable clusters than in measuring their ability on some performance task. However, experimental comparisons suggest that these methods can produce as accurate knowledge structures as the divisive algorithms we described in previous sections.

7.5 Variations on hierarchy formation

Now that we have considered some basic approaches to the formation of concept hierarchies, we can consider some refinements that cut across the methods. In this section we examine a variety of extensions, including alternative search-control schemes, extensions to more complex memory organizations, and methods for transforming concept hierarchies into other structures.

7.5.1 Alternative search algorithms

The algorithms we have presented in this chapter have an important feature in common – they carry out greedy or hill-climbing searches through the space of concept hierarchies. Our bias toward such methods reflects a strong trend in the literature, but this does not mean that more sophisticated, search-intensive methods are not possible. For instance, one might retain multiple competing hierarchies rather than only one, pursuing a beam search that uses the same learning operators and evaluation metrics.

A less obvious extension involves lookahead search in the hierarchy space. For example, in constructing a univariate decision tree, the attribute that appears most discriminating at the current level might lead to clusters that cannot easily be distinguished further. In contrast, a second attribute might look less informative in isolation, but lead to excellent distinctions when combined with a second feature. A revised version of DDT that considers all *pairs* of attributes, which equates to looking two steps ahead in the search, will detect such situations. Deeper lookaheads will be more expensive but can find more complex interactions among attributes. An extreme version of this approach carries out best-first search through the space of concept hierarchies.

Another scheme uses a greedy method to efficiently construct an initial concept hierarchy, then applies hierarchy-altering operators in the hope of producing better structures. This approach requires an evaluation metric over the entire hierarchy, which it can use to decide whether a revised structure is preferable to the current one. This method constitutes a form of 'anytime learning', in that it continues running as long as computational resources are available, with longer runs tending to produce better hierarchies, at least on the training set.

7.5.2 Forming more complex structures

Although most research on hierarchy formation has dealt with tree structures, one can imagine domains in which more complex organizations of memory would be useful. For example, a univariate decision tree requires $2^{k+1}-1$ nodes to represent a k-bit parity concept, in which an instance is positive if an even number of the k relevant Boolean features are true and negative otherwise. In contrast, one can represent the same target concept with only $2k+1$ nodes, provided that one uses a directed acyclic graph (DAG) rather than a tree.

The DDT algorithm requires two modifications to construct such 'decision DAGs'. First, one must include an operator for merging two nodes into a single node, which is needed to obtain nodes with multiple parents. Second, one must use an evaluation metric that, in some cases, prefers a DAG structure to the equivalent but more complex tree structure. Notions of minimum description length (Chapter 5) seem well suited for this purpose. In addition, DDT constructs decision trees in a depth-first fashion, expanding nodes on the left before expanding with ones further to the right. Formation of decision DAGs is aided by a more flexible control structure that expands whichever node best improves the quality of the hierarchy.

In unsupervised learning, the performance task can involve prediction of multiple features. In some domains, the best hierarchy for predicting one feature may differ from the best hierarchy for predicting another. For example, suppose some animals are carnivores, whereas others are herbivores; also suppose that some occur in a zoo and others in homes. One might well want to organize the instances into two interleaved trees, the first related to eating habits and the second to surroundings, with the predictable and predictive features differing in the two trees. One response to this situation is to formulate a hierarchy that interleaves OR nodes, which sort instances down one branch or another, with AND nodes, which sort instances down all branches.

One can adapt both divisive and agglomerative approaches to the construction of such hierarchies. For example, in generating a univariate concept hierarchy, one might select the k most discriminating attributes rather than only one, and thus produce a tree with k complementary AND nodes at each choice point, alternating with OR nodes. A similar approach applies to the divisive formation of multivariate concept hierarchies. The challenge here involves deciding on the appropriate number of AND children for a given OR node, just as one must decide on the number of OR children in simpler multivariate tree structures. In the agglomerative framework, one can select the k nearest neighbors at each level, creating multiple interleaved trees from the bottom up that organize instances in complementary ways.

7.6 Transforming hierarchies into other structures

As we have seen, concept hierarchies impose a particular organization on memory. This structure seems quite different from those we considered in Chapters 5 and 6, but there are special cases in which there exists a clear mapping from one type of organization to another. For example, univariate decision trees are a strict subset of decision lists with logical tests, and there is a simple method that transforms the former structures into the latter.

Table 7-8 presents pseudocode for DTL (decision tree to decision list), an algorithm that carries out this transformation. Given a univariate decision tree, the method traverses each path through the tree, collecting tests along the way. Upon reaching each terminal node, it creates a logical rule using the tests it has collected in the left-hand side and

Table 7-8. The DTL algorithm for transforming a univariate decision tree into a decision list.

```
Inputs: The current node N of the decision tree.
        The RULES collected so far.
        The conditions LHS collected for the current rule.
Output: A logical decision list.
Top-level call:  DTL(root, { }, { })

Procedure DTL(N, RULES, LHS)
 If N is a terminal node,
    Then let C be the class stored at N.
         Insert the rule 'If LHS, then C' into RULES, using the
            percentage of C instances at N to determine order.
         Return RULES.
    Else for each child S of node N,
         Let T be the attribute-value test for node S.
         Let LHS′ be the union of T and LHS.
         Let RULES be DTL(S, RULES, LHS′).
```

the class for the terminal node in the right-hand side. DTL uses the accuracy of each terminal node on the training set (the percentage of instances with the specified class) to order rules in the decision list.

One cannot turn any multivariate decision tree into a decision list, but such a tree can be transformed into a special form of inference network. Consider a binary multivariate tree in which each pair of siblings is discriminated by a linear threshold unit. We can redraw such a tree as a three-layer inference network. Each internal node from the tree becomes a node in the first layer, each *path* through the tree produces a conjunctive node in the second layer (with inputs from each internal node along the path), and each class produces a disjunctive node in the third layer (with inputs from each path that predicts that class). Similar mappings exist for multivariate trees that organize logical and competitive descriptions, although the former produces two levels of conjunctive nodes that could be combined into a single layer.

The above treatment suggests that concept hierarchies are simply special cases of decision lists and inference networks. However, our examples have focused on decision trees, in which a single predictable

attribute (the class) is stored with terminal nodes. The relationship is less clear for concept hierarchies that store multiple predictable attributes (sometimes at internal nodes), and for hierarchies that alternate between AND and OR nodes or that organize concepts into directed acyclic graphs.

7.7 Summary of the chapter

In this chapter we examined concept hierarchies as an organization for memory, along with the use and acquisition of such structures. We found that, like inference networks, concept hierarchies can organize logical, threshold, or competitive concepts, but that they work in a quite different manner. In general, the performance component sorts a new instance downward through the hierarchy, at each choice point using predictive features to select a branch, and using predictable features to make inferences when the sorting process halts.

Although performance always takes place from the top down, we found that the formation of hierarchies can occur in either a divisive or agglomerative fashion. We encountered three nonincremental algorithms that operate divisively: the DDT method clusters training cases into groups based on a single attribute's values and creates univariate trees; DMT forms clusters based on instances' class and constructs multivariate hierarchies; and iterative optimization uses a more sophisticated hill-climbing process to cluster the data.

We also saw two divisive algorithms that operate incrementally, thus integrating performance with learning. The first, HDT, retains statistics for predictive attributes with each node that let it revise a univariate tree in the light of new data. The second, IMT, stores similar information for use in extending or revising multivariate concept hierarchies. Both methods rely on bidirectional learning operators to minimize the effects of training order.

Finally, we considered ACH, an agglomerative method that constructs concept hierarchies from the bottom up. This algorithm computes the distance between all pairs of training instances, selects the closest for merging, replaces the pair with their parent, and repeats the process. Because agglomerative methods reconsider many instances on each iteration, they are typically nonincremental. Research on these techniques has used a variety of distance metrics that can produce quite different hierarchies for the same data.

Most research on the formation of concept hierarchies has focused on tree construction using greedy algorithms, but similar methods can organize concepts into more complex structures such as directed acyclic graphs, and more sophisticated search schemes are also possible. Both types of extension require search through an expanded space, but in some cases they can produce hierarchies that are both simpler and more accurate on novel test cases. We also saw that one can transform decision trees into decision lists or inference networks, if one prefers such organizations, but that this does not necessarily hold for arbitrary concept hierarchies.

Exercises

1. Show a univariate decision tree that the DDT algorithm (Section 7.2.1) constructs from the training data in Figure 5-4 (from Chapter 5), along with the evaluation scores for each attribute selected. Break ties among attributes randomly. If a tie occurs at the top level, show a second decision tree based on selecting a different split.
2. Draw a univariate decision tree that DDT constructs for the numeric data in Figure 5-2 (from Chapter 5). Also show the decision regions that this tree produces, labeling each boundary with the order in which the algorithm introduces it. Repeat this process for the training data in Figure 5-7.
3. Show the multivariate decision tree that the DMT algorithm (Section 7.2.3) generates for the cases in Figure 5-2, using an instance-averaging subroutine (Chapter 4). Show the mean values for the attributes at each node in the hierarchy, along with the decision boundaries they produce. Do the same for the data in Figure 5-7.
4. Ignore the classes associated with instances in Figure 5-7 and apply k-means clustering, with $k = 2$, to divide the data into two groups. Select the bottom left and far right instances as initial seeds, and show both the clusters and their means on each iteration of the method until it converges. Repeat the process with the two leftmost instances as seeds.
5. Trace the IDT algorithm (Section 7.3.1) on the training data in Figure 5-5 (from Chapter 5), showing which instances lead to changes in the decision tree and the structure that results in each case. Do the same for a similar training set in which the fourth instance (d) is negative rather than positive.

6. Trace the behavior of the HDT algorithm (Section 7.3.2) on the training data from Figure 5-4, showing the scores of each attribute at the affected node whenever the tree structure changes. Assume that the method initially sees the first two positive instances, followed by the first negative case, after which instances of the two classes alternate. Show the tree that results if one runs the data through twice in the same order.
7. Trace the behavior of the IMT algorithm (Section 7.3.3) on the training data in Figure 5-8 (from Chapter 5). Use Euclidean distance of an instance from a node's mean as the latter's score, and assume that each instance is always sorted to a terminal node (i.e., ignore node merging, splitting, and creation at nonterminal levels). Repeat this process on the same data, but prefer node creation when the Euclidean distance is greater than 5 from the nearest node's mean.
8. Show the distance matrix for the five instances from Exercise 7, using Euclidean distance, along with the revised matrix after the ACH algorithm (Section 7.4) creates the first cluster. Draw the structure of the hierarchy that ACH generates, and specify the mean values for each node in the hierarchy.
9. Recall that the data summarized by the decision tree in Exercise 1 can also be described by the DNF formula in Figure 5-4 (d). Draw a decision DAG (Section 7.5.2) that is equivalent to this formula, without specifying the algorithm used to construct it. Compare the number of nodes in this DAG to the number in the tree from Exercise 1.
10. Use the DTL algorithm (Section 7.6) to transform the decision tree from Exercise 1 into a decision list. Can you remove any conditions from the resulting rules without altering the decision list's behavior on the training data?

Historical and bibliographical remarks

Univariate concept hierarchies (simple decision trees and discrimination networks) have played a central role in machine learning since its earliest days. Hunt (1962) describes CLS, an early version of the nonincremental DDT algorithm, which he proposed as a model of human learning; Hunt, Marin, and Stone (1966) built on this approach and presented the first extensive experimental study of concept induction. Quinlan's (1983) ID3 extended the CLS framework to incorporate a decision-theoretic evaluation metric and other features; his system has been used by many researchers in comparative studies but no longer has status as a psychological theory. Its successor, C4.5 (Quinlan, 1993b), is also widely used by induction researchers; it includes a number of improvements, including use of the DTL algorithm (due to Quinlan, 1987) to translate a decision tree into a decision list. Mingers (1989a) reviews many of the evaluation metrics used in decision-tree induction and reports an experimental comparison, as do Buntine and Niblett (1992).

Independent but related lines of research come from the social sciences and statistics. In the former community, Sonquist, Baker, and Morgan (1977) proposed AID, a method for univariate decision-tree induction that uses a statistical measure to select discriminating attributes. Biggs, de Ville, and Suen (1991) review this approach and various extensions, broadly referred to as CHAID. In statistics, Friedman (1977) described another variant of the DDT algorithm for creating decision trees, which Breiman, Friedman, Olshen, and Stone (1984) developed into CART, a software package that has been widely used in statistics and, to a lesser extent, in machine learning.

Much of the recent research has explored variations and extensions of the basic DDT algorithm. Patterson and Niblett (1982) and Breiman et al. (1984) describe approaches to creating binary splits from numeric attributes, Cestnik, Kononenko, and Bratko's (1987) ASSISTANT constructs binary trees that contain arbitrary subsets of nominal values, and Breiman et al.'s CART creates trees with n branches, each with an arbitrary subset of nominal values. CART was also the first system to induce multivariate decision trees, using the IWP algorithm from Chapter 3 as a subroutine. More recently, Utgoff and Brodley's (1990) LMDT method induces multivariate trees composed of linear machines, Murthy, Kasif, Salzberg, and Beigel's (1993) OC1 uses a variation on the CART scheme, and Langley's (1993) recursive Bayesian classifier uses the DMT algorithm (Section 7.2.3) to form a hierarchy of probabilistic descriptions.

Research on the incremental formation of concept hierarchies also has a long history. The ICH algorithm in Section 7.3 is based on Feigenbaum's (1961) EPAM, a model of human memorization and retrieval that has had a pervasive influence on both psychology and machine learning. Richman (1991) reviews some recent extensions to EPAM and the system's psychological status. Lebowitz's (1987) UNIMEM revised the EPAM approach to handle multivariate concept hierarchies, and Fisher's (1987) COBWEB (the basis of our IMT algorithm) refined this approach and provided a probabilistic semantics. Hadzikadic and Yun (1989), McKusick and Langley (1991), and Martin and Billman (1994) have also done work in this tradition. A collection by Fisher, Pazzani, and Langley (1991) contains a representative sample of work on the incremental induction of concept hierarchies. The HDT algorithm for incremental induction of decision trees, which represents a somewhat different line of research, comes from Schlimmer and Fisher's (1986) ID4, which Utgoff (1989) extends in his ID5R system.

Work on agglomerative methods like the ACH algorithm has been less common within the machine learning community, though Hanson and Bauer's (1989) WITT is one exception. Agglomerative techniques are more widely used within numerical taxonomy; Annenberg (1973) and Everitt (1980) provide broad reviews of this area, while Fisher and Langley (1986) discuss its relation to machine learning work on hierarchy formation. Techniques for iterative optimization like the k means and EM algorithms are also widely used in numerical taxonomy, but some work (e.g., Michalski & Stepp, 1983; Jordan & Jacobs, 1993) also appears in the machine learning literature on hierarchy construction.

A few researchers have explored variations on the basic methods for forming concept hierarchies, as outlined in Section 7.5. For example, Norton (1989) describes an extension of the DDT algorithm that uses multi-step lookahead. Oliver (1993) reports a version of DDT for inducing univariate decision DAGs, whereas Kohavi (1994) presents a quite different approach to the same task. Martin and Billman (1994) describe a variation of the IMT method that creates concept hierarchies with interleaved AND and OR nodes. Our DTL algorithm for translating decision trees into decision lists borrows from Quinlan's (1987, 1993b) work in this area.