However, work in this framework has entered the machine learning literature only in recent years, starting in the late 1980s. Nor are the simple instance-averaging and attribute-weighting methods in Sections 4.1.2 and 4.1.3 representative of work in this area. We used them only to introduce the ideas of storing prototypes and giving their features different weights. We discuss more realistic techniques for instance-based learning, such as nearest neighbor algorithms, in the next chapter.

One might expect the same comment to hold for the naive Bayesian classifier, given its simplicity. Indeed, although first introduced by Good in 1950, this method originally appeared in the literature on machine learning as a straw man against which to compare much more sophisticated induction methods (e.g., Clark & Niblett, 1989). However, its impressive performance in many domains has led some researchers to study the Bayesian classifier in its own right (Langley, Iba, & Thompson, 1992) and others to explore extensions of the method (Kononenko, 1991; Langley & Sage, 1994a; Pazzani, 1995).

CHAPTER 5

The Construction of Decision Lists

The previous chapters dealt with the induction of concepts that one could describe using a single decision region. This focus was useful for historical reasons, and it let us introduce some basic methods for induction, but such simple learning methods seem unlikely to be useful for practical applications. In this chapter we turn to the induction of *disjunctive* descriptions, which provides one approach to creating multiple decision boundaries, and we will examine alternative schemes in the following chapters. However, we will also find that the more sophisticated methods build on the simpler ones we have already seen.

We start the chapter by considering generic issues in the representation, use, and induction of disjunctive descriptions. After this we present a number of algorithms for disjunctive induction that vary in their techniques for partitioning the training data and organizing terms in the concept description. In general, we focus on heuristic approaches rather than exhaustive ones, and we discuss incremental methods only after describing the more basic nonincremental ones.

5.1 General issues in disjunctive concept induction

As before, we initiate our discussion with some issues that go beyond particular algorithms for the induction of disjunctive concept descriptions. Below we consider data structures and interpretation schemes for representing and using such induced knowledge. After this, we attempt a more formal statement of the task of disjunctive induction.

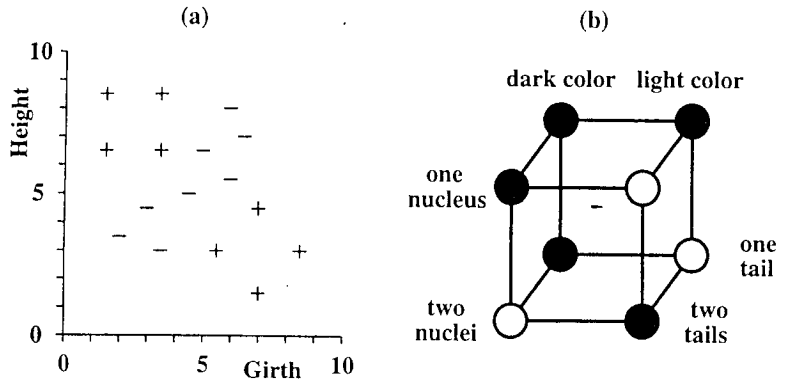

Figure 5-1. Positive and negative instances from (a) the numeric height/girth domain and (b) the three-attribute cell domain that one cannot distinguish using a single decision region. Such training sets suggest the use of representations that involve disjunctive descriptions.

5.1.1 The representation and use of disjunctive descriptions

Consider the data in Figure 5-1, which presents positive and negative instances from (a) the height/girth domain and (b) the three-attribute cell domain. Although the attributes are familiar, the distributions of the instances are quite different from those we have seen before. In neither case can one draw a single decision region that partitions the positive instances from the negatives. This characteristic is completely separate from the representational formalism one uses to express the decision boundaries, whether the logical conjunctions we encountered in Chapter 2, the threshold units we saw in Chapter 3, or the competitive descriptions we discussed in Chapter 4.

Clearly, these induction problems require some more powerful language that lets one express more than one decision region. In this chapter we will explore three alternative ways to achieve this effect. The first, which can be used to combine either logical or threshold concepts, is known as *disjunctive normal form* (DNF). This scheme combines a set of simple descriptions, D_1, D_2, ..., D_n, into the logical disjunction $\{D_1 \vee D_2 \vee \ldots D_n\}$. If any of the simple descriptions matches an instance, then one classifies the instance as a member of the concept. We will refer to each simple description within a DNF expression as a *term*.

A drawback of DNF notation is that, when extended to multi-class prediction, the expressions for more than one class can match an instance. One solution, which we will not consider in detail, is to ensure that expressions for each class are mutually exclusive. Another response is to place an ordering on the terms, so that if two or more simple descriptions match an instance, the one with precedence wins the conflict. Such an organization of knowledge, known as a *decision list*, has rather different characteristics than disjunctive normal form. A decision list typically includes a default term that occurs last but always matches, leading its associated class to be predicted if no other term applies to an instance.

Using disjunctions of rectangles and spheres, one can approximate any possible target concept, whether it involves Boolean, nominal, or numeric attributes. Disjunctions of linear threshold units can also represent any Boolean concept, but in numeric domains, the one-sided nature of linear threshold units limits the concepts they can represent. For this reason, our examples of threshold concepts in numeric domains will emphasize spherical units rather than linear ones.

Both the DNF and decision-list schemes make sense for logical and threshold descriptions, but not for competitive concepts, since the latter cannot be understood in isolation. However, there also exists a natural extension of competitive concepts to multiple decision boundaries. One simply allows two or more descriptions for each class, then selects the best of these during the match process. As before, the decision boundaries emerge from the entire set of descriptions, but some of these descriptions (and their associated regions) happen to predict the same class. We will refer to this representational scheme as a *competitive disjunction*.

The storage of multiple descriptions for each class suggests the idea of taking them all into account during the classification process. This scheme makes special sense within a competitive framework, and a variety of methods for voting and otherwise combining evidence have been developed within both the instance-based and probabilistic frameworks. We will not focus on such performance methods here, but we will return to them in Chapter 9, when we examine their use in handling noise.

5.1.2 The task of disjunctive induction

Now that we have described the basic representation and performance assumptions of disjunctive concepts, we can clearly state the task of inducing them from experience:

- *Given*: A set of training instances, each with associated class labels;
- *Find*: A disjunctive description that, to the extent possible, correctly classifies novel test instances.

Thus, methods that address this induction task must accept as input a set of classified instances, and they must generate a DNF expression, decision list, or competitive disjunction for use in classifying future instances. This task statement is very similar in form to the statements for simple induction that we presented in previous chapters. The only difference lies in the 'find' clause, which lets each class be associated with more than one simple description.

As a result, algorithms that address this task must search a space of disjunctive descriptions, rather than the smaller space of simple conjunctive, threshold, or competitive descriptions. Not surprisingly, this task is more difficult than the ones we considered in Chapters 2, 3, and 4. At least for some representations, the space of disjunctive descriptions is partially ordered according to generality, but the branching factor is so large that in practice no researchers have attempted to use this structure in any systematic way. Instead, we will see shortly that many approaches to disjunctive induction deal with this issue by partitioning the task into a set of more tractable component induction tasks.

Our discussion of disjunctive induction will focus on nonincremental methods, although we will see some incremental variants. We will also emphasize the two-class case, although we discuss some extensions to multi-class prediction. The techniques we will examine draw on the algorithms from earlier chapters, calling on them as subroutines to generate the descriptions of which the disjuncts are composed. Our examples will typically ignore the details of these subroutines' behavior and examine only the state of knowledge before and after their application. We will also emphasize heuristic subroutines, since methods that insist on perfect accuracy over the training set will fail when their assumptions are violated, as in disjunctive domains.

Table 5-1. The NSC algorithm: Nonincremental induction of DNF expressions using a separate-and-conquer approach.

```
Inputs: PSET is the set of positive training instances.
        NSET is the set of negative training instances.
        DNF is a disjunction of single-region descriptions.
Output: A disjunction of single-region descriptions.

Top-level call: NSC(PSET, NSET, { })

Procedure NSC(PSET, NSET, DNF)

If PSET is empty,
Then return DNF.
Else find a single-region description D that covers some
        instances in PSET but covers no instances in NSET.
     Add the description D to DNF.
     Remove the instances covered by D from PSET.
     Return NSC(PSET, NSET, DNF).
```

5.2 Nonincremental learning using separate and conquer

Let us start by examining the induction of descriptions stated in disjunctive normal form, for use in discriminating between two classes. One can reduce this task to the construction of a DNF expression for one of the classes and use the other class as a default when the expression fails to match. Technically, this makes the result a decision list, but only a degenerate one.

Table 5-1 presents pseudocode for a nonincremental separate-and-conquer (NSC) algorithm that addresses this learning task. The method accepts a set of positive training instances (for the preferred class), a set of negative instances (for the default class), and a DNF expression composed of disjuncts of terms, which is initially empty. NSC invokes an inductive routine to construct a description that covers some of the positive instances but none of the negative instances. Many of the logical or threshold methods from previous chapters can serve in this step, although some must be modified to favor exclusion of negative instances over coverage of positive ones.

NSC adds the resulting description to its DNF expression, puts aside the instances covered by this term, and calls on itself recursively with

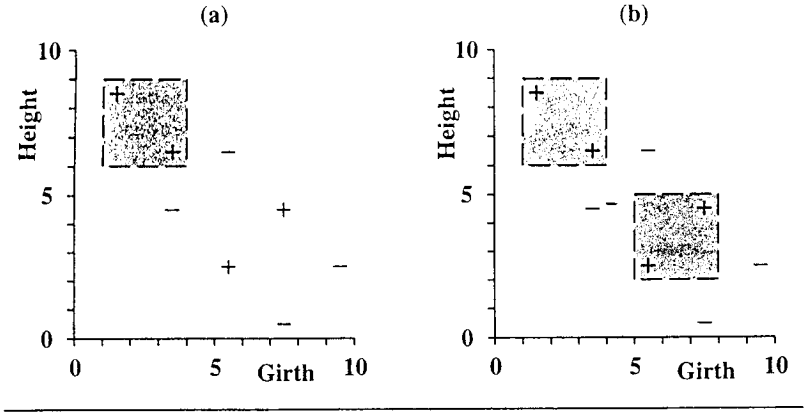

Figure 5-2. Behavior of the NSC method on training instances in the height/girth domain (a) after one call to the HSG subroutine, and (b) after a second call to the same algorithm.

this reduced training set and extended expression. If no more positive instances remain, the algorithm returns the current DNF expression and halts. Otherwise it repeats the above process until it finds an expression that covers all positive training cases but none of the negatives.

Some examples of NSC's operation should clarify its interaction with the methods from earlier chapters. Let us first consider the effects when NSC uses the HSG algorithm from Chapter 2 to determine each term in the DNF expression. Recall that HSG carries out a beam search from specific to general descriptions, using an evaluation function to direct the search and returning a set of descriptions with acceptable scores. Here we assume that HSG continues to generate more general descriptions until their evaluation scores cease to improve, then selects the best description and halts.

Figure 5-2 depicts four positive and four negative instances from the numeric domain involving `height` and `girth`. The layout makes it clear that one cannot draw any rectangle around the positive cases that does not also include some negatives. Thus, the simple conjunctive descriptions that we assumed in Chapter 2 will not suffice in this situation, and we must turn to a more powerful description language. Given these training instances, NSC's first action is to call the subroutine HSG, which immediately selects one of the positive cases as a seed, say the instance `height 8.5, girth 1.5`. Its beam search then considers progressively more general versions of this initial conjunction, eventually producing the rectangular decision region shown in Figure 5-2 (a).

No further generalizations are possible without covering negative cases, so NSC removes the two covered instances and calls HSG again on the remaining training data. The subroutine selects another seed, let us say the instance `height 2.5, girth 5.5`. After carrying out a second beam search, the algorithm finds a second conjunction that covers an additional two positive instances but no negatives. NSC adds this description to its DNF expression, giving the extensional definition shown in Figure 5-2 (b). Since this covers all the positive but none of the negative instances, the algorithm halts, returning the expression `((< height 9) ∧ (< girth 4) ∧ (> height 6) ∧ (> girth 1)) ∨ ((< height 5) ∧ (< girth 8) ∧ (> height 2) ∧ (> girth 5))` as its result.

Clearly, the NSC algorithm is not wedded to any particular routine for inducing the terms of its DNF expression. For example, one can easily replace the call to HSG with one to HGS, the method for general-to-specific beam search described in Chapter 2. This scheme also produces a disjunction of conjunctive terms, but the details will differ because of the two routines' different inductive biases. Figure 5-3 (a) clarifies this point by showing the decision boundaries that result when NSC uses the HGS routine to identify terms. Because this method prefers general descriptions to specific ones, the final DNF expression covers a larger fraction of the instance space than when HSG is used, and thus would classify more test cases as positive.

One can also use techniques for inducing threshold concepts to construct the terms in the DNF expression. The basic methods tend to give equal emphasis (and thus equal weight changes) to misclassifications of positive and negative training instances. In disjunctive domains, this leads them to produce decision boundaries that cover some positive and some negative cases. However, a simple alteration of the update functions can bias search toward threshold units that exclude negative instances while covering some but not all positives. Of course, one need not insist that each term in the DNF expression rule out all of the negative instances, and covering some negatives may even be desirable in noisy domains, but clearly the expression should cover primarily positive instances. Figure 5-3 (b) shows the result when NSC uses a modified technique for inducing spherical threshold units.

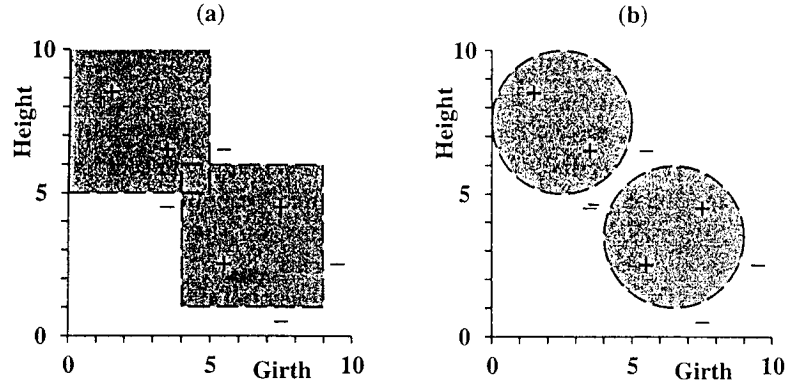

Figure 5-3. Decision boundaries produced by the NSC algorithm (a) when using the HGS routine to produce terms in the DNF expression, and (b) when invoking a modified technique for spherical threshold units.

Now let us consider a second data set from the four-attribute cell domain. Figure 5-4 (a) presents six positive instances and (b) shows four negative instances from this domain that one cannot separate with a single decision boundary. Suppose we run NSC on these data, using the HSG algorithm to generate terms in the disjunction, combined with the metric $(P_c + N_{\bar{c}})/(P + N)$ to evaluate individual conjunctions.

In this situation, the first call to HSG produces the logical conjunction `two tails` $\wedge$ `two nuclei`, which covers the first four positive instances in the figure but none of the negatives. When NSC removes these four cases from the data set and calls HSG again, the latter finds the conjunction `one tail` $\wedge$ `dark color` $\wedge$ `thick wall`, which covers the last two positives but without covering negative instances. Since no positive cases remain, NSC halts and returns the disjunction of the two terms, which Figure 5-4 (c) depicts graphically.

Similar results occur when NSC uses the HGS algorithm, combined with the same evaluation metric, to find the component terms. Although this subroutine searches from general to specific descriptions, it finds the same two-attribute conjunction to cover the first four positive instances as does HSG. However, when called on the remaining two positive cases, its different search bias leads to the more general conjunction `dark color` $\wedge$ `thick wall`. Again, NSC halts at this point,

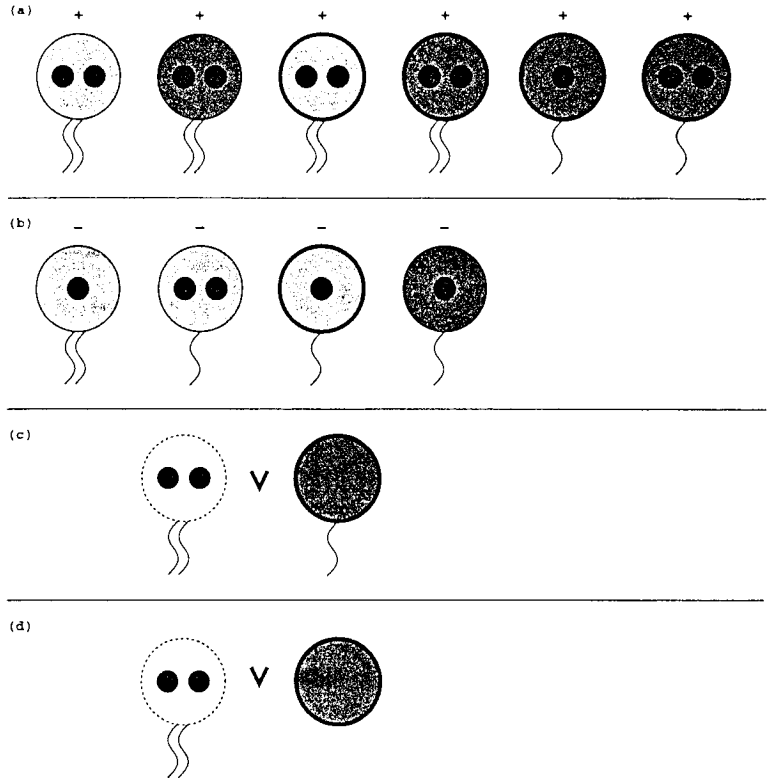

Figure 5-4. Behavior of the NSC algorithm on data from the four-attribute cell domain, given (a) six positive instances that fall into two separate decision regions and (b) four negative instances. When using HSG as a subroutine, NSC produces logical disjunction (c), whereas HGS leads to disjunction (d).

having completely separated the two classes with the disjunction shown in Figure 5-4 (d). Although the descriptions in (c) and (d) both handle the training set, they make different predictions about unseen instances, as would the DNF expressions that NSC would produce if instead it used subroutines for inducing threshold units.

Like the algorithms from previous chapters, the NSC method is designed to induce DNF expressions for a single class. However, one can use NSC as the building block for a more sophisticated scheme that

Table 5-2. The MSC algorithm: Nonincremental induction of multi-class decision lists using separate and conquer.

```
Inputs: CSET is a set of two or more class names.
        ISET is a set of classified training instances.
Output: A decision list composed of single-region descriptions.

Procedure MSC(CSET, ISET)

Let RULE-SET be the empty set.
For each CLASS in CSET,
    Let PSET be instances in ISET labeled with CLASS.
    Let NSET be instances in ISET not labeled with CLASS.
    Let DNF be NSC(PSET, NSET, { }).
    For each term D in DNF,
        Add the rule 'If D, then CLASS' to RULE-SET.
Eliminate possible conflicts among descriptions
   in RULE-SET that predict different classes.
Return RULE-SET.
```

handles domains involving more than two classes. Table 5-2 presents pseudocode for MSC, a separate-and-conquer algorithm designed for the task of inducing multi-class decision lists.

The input to MSC is a set of class names and a set of classified training instances. For each class, the method collects the set of training cases with that class name, as well as a set of those with other associated classes. The former become positive instances and the latter negative instances, which MSC passes on to the NSC subroutine. This returns a DNF expression, which MSC converts into an ordered set of decision rules that predict the class name. The algorithm repeats this entire process for each of the classes.

However, two problems arise with the generation of separate DNF expressions for each class. First, some regions of the instance space may remain uncovered by any expression, but one can easily correct this matter by adding a default rule. Second, some regions may be covered by more than one expression, leading to conflicts on certain test cases. Insistence on a pure DNF notation requires that one somehow ensure that the expressions for different classes have no overlap. In some cases, this can be accomplished by making some terms in the DNF expression more specific, but in other cases it requires that one decompose the expression into more terms. Since few induction methods go to such lengths, we will not consider them further.

A more elegant response simply places an ordering on the classes based on their relative frequencies. This approach transforms the DNF expression into a decision list, though one in which all terms for a given class occur together. Alternatively, one can place an ordering on each term separately, based on the number of training instances covered by each one. This scheme (combined with a default rule) produces a fully general decision list.

A related approach interleaves the term-finding process among the classes. First one uses a simple induction algorithm to find a logical or threshold expression for the most frequent class. The resulting description becomes the first term in the decision list, and one removes the instances that it covers. One then repeats the process for the class that occurs most often among the remaining instances, generating the second term in the decision list. This process continues until the remaining instances reside in a single class, which becomes the default. The interleaved decision list produced by this method may make different predictions than those generated by other techniques, but their relative performance on a given domain remains an open question.

5.3 Incremental induction using separate and conquer

In earlier chapters we argued that incremental hill climbing constitutes an important approach to learning, and we saw some uses of this idea in the induction of simple concepts. Table 5-3 presents pseudocode for ISC, an incremental separate-and-conquer version of the NSC algorithm that uses the same basic search control as the IHC scheme from Chapter 2. Like its relative, ISC retains a single hypothesis in memory, but in this case the concept description is not a simple conjunction but a disjunctive set of logical or threshold terms, which it initializes to a single term at the outset. The algorithm also retains up to the most recent k training instances for use in evaluating alternative hypotheses and, like IHC, it revises its hypothesis only when it makes an error in classification.

If ISC mistakenly classifies a positive instance as negative, it takes immediate steps to produce a more general hypothesis. There are two natural ways to accomplish this goal. The algorithm can modify a term

Table 5-3. The ISC algorithm: Incremental hill-climbing search for a DNF expression using separate and conquer.

```
Inputs: ISET is a set of classified training instances.
Output: A disjunction of single-region descriptions.
Parameters: K is the number of instances retained in memory.

Procedure ISC(ISET)

Let H be a set containing a maximally specific single-region
    description based on the first positive instance in ISET.
For each misclassified instance I in ISET,
    Let KSET be the previous K (or fewer) instances.
    If the training instance I is positive,
    Then for each term C in hypothesis H,
             Let S be the minimal revision of C that matches I.
             Generate a revision of H in which S replaces C.
         Let BEST be the best-scoring of these revisions.
         Let NEW be Add-Term(H, I).
    Else if the training instance I is negative,
         Then let BEST be a copy of H.
              For each term C in BEST that matches I,
                  Let G be a minimal revision of C that
                      does not match the instance I.
                  Replace C with G in the hypothesis BEST.
              Let NEW be Remove-Term(H, C).
    If Score(NEW, KSET) > Score(BEST, KSET),
    Then let BEST be NEW.
    If Score(BEST, KSET) > Score(H, KSET),
    Then let H be BEST.
Return the hypothesis H.

Note: Add-Term adds a new single-region description to the
      hypothesis H based on the positive instance I.
```

in its DNF expression just enough to let it match the training case. This can involve removing a test for Boolean, nominal, or numeric features, increasing the size of a rectangle, or changing the weights on a threshold unit. Because the disjunctive hypothesis may contain multiple terms, ISC must apply this generalization process to each one. At a minimum, this produces one modified description for each term.

Another alternative, not available in simpler tasks, involves adding an entirely new term, using the subroutine `Add-Term` in the table. For the resulting hypothesis to match the training instance, only one of the DNF terms must match it. The simplest way to ensure this is to add a term that matches only the positive instance itself. Another scheme would generate the most general clause that matches none of the stored negative instances; this would produce a more general hypothesis, but it also has a less incremental flavor than the other approach.

The ISC algorithm responds differently when it mistakenly classifies a negative instance as positive. Again, there are two obvious approaches to correcting the overly general hypothesis. One can modify each term matching the instance just enough to prevent the match. Depending on the representation for terms, this can involve adding a Boolean, nominal, or numeric feature, shrinking a rectangle, or modifying weights in a threshold unit. The second alternative is to eliminate an overly general term from the hypothesis entirely.

Once it has constructed the various generalizations (or specializations) of its current hypothesis, ISC uses an evaluation function to select one over the others. If the disjunctive description with a new (or removed) term has the best score, it replaces the existing description. Similarly, if one of the descriptions containing a generalized (or specialized) term appears best, it becomes the new hypothesis. For some evaluation functions, the existing description may score better than the proposed successors despite its error on the latest training instance, in which case it is retained unaltered.

However, the incremental nature of the ISC algorithm makes greater demands on the evaluation metric than does the NSC method. The latter has an implicit bias toward expressions with fewer terms, since its considers adding new terms only if the existing disjunction fails to cover some positive instances. ISC only considers adding a new term when it makes a similar error, but its incremental character can let this occur before the algorithm sufficiently masters other terms.

As a result, ISC must decide between modifying its existing terms in the disjunction and adding or deleting terms. In principle, the algorithm could simply create a disjunction composed of each positive instance it encounters, but this is not what we typically desire from an induction method. We need some way of introducing search bias into ISC's learning behavior. The most natural approach is to include some measure

of an expression's simplicity in the evaluation function used to direct search. For instance, one might measure simplicity as $\frac{1}{t}$, where t is the number of terms in a disjunct. However, we must still combine this with some measure of the expression's accuracy a on the last k training cases, such as $a = (P_c + N_{\bar{c}})/k$, where P_c is the number of positive instances covered and $N_{\bar{c}}$ is the number of negatives not covered.

One obvious evaluation metric is simply to take the sum of these two quantities, $a + \frac{1}{t}$, which will range from 0 to 2. A more sophisticated scheme introduces a tradeoff parameter ω that specifies the relative importance of these two factors, giving the metric $(1-\omega)\cdot a+\omega\cdot\frac{1}{t}$, which ranges between 0 and 1. A more principled approach would use the notion of *minimum description length*, an information theoretic measure that reports the total amount of information needed to describe all the training data. Briefly, this encodes both the summary description and the incorrectly classified instances in terms of the bits it would take to communicate them, providing a uniform way to measure the tradeoff in complexity and coverage.

Figure 5-5 traces ISC's behavior on a training set from the four-attribute cell domain, assuming a logical representation for component terms, use of the evaluation function $a + \frac{1}{t}$ as described above, and retention of the last four training instances in memory. Training instances appear on the left in the order processed, and the resulting DNF expression appears on the right, along with the scores for the two components of the evaluation metric and their sum.

Item (a) shows the initial training case and the one-term disjunction that ISC produces in response: `one nucleus`, `two tails`, `thin wall`, and `light color`. Items (b), (d), and (f) illustrate positive instances that lead to more general terms in the DNF expression, whereas (c) shows a negative instance that produces a more specific term. Item (e) involves a positive instance that causes the algorithm to introduce a second term into the disjunctive hypothesis. The figure shows only the best-scoring DNF expression in each case, not the alternatives that ISC generates during its search.

As we noted in Chapter 2, induction methods that employ incremental hill climbing have low requirements for both memory and processing. They consider only one hypothesis, and they store and reexamine a constant number of training instances at each step during learning. As before, the price is a sensitivity to the order of training instances and,

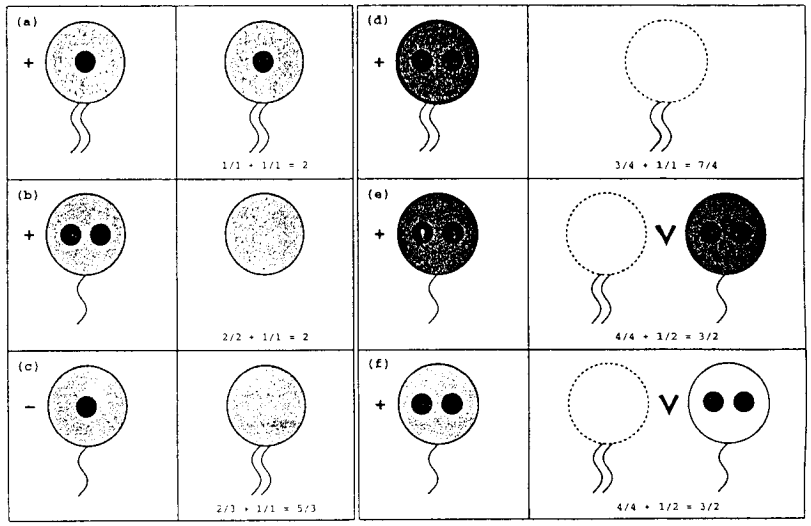

Figure 5-5. Behavioral trace of the ISC algorithm in the four-attribute cell domain, with (a) representing the first instance and DNF hypothesis and (f) indicating the sixth instance and hypothesis. Instances appear on the left, the resulting hypotheses appear on the right, and the score of the evaluation function occurs beneath the latter.

typically, a reliance on more training data to arrive at the target concept than for nonincremental heuristic methods. Moreover, algorithms like ISC are not guaranteed to converge, and in noisy domains they may change their hypothesis even if they have already reached the target concept. However, these characteristics also hold for another class of incremental learners, human beings. Both also tend to favor concept descriptions with small numbers of terms, although they can entertain more complex ones when necessary.

5.4 Induction of decision lists through exceptions

The separate-and-conquer method described above constructs a decision list in top-down order, adding the first term in the list in its first step, then the second, and so forth. But clearly one can operate in the opposite direction as well, and Table 5-4 summarizes NEX (nonincremental induction using exceptions), an algorithm that induces a decision list in this way. The resulting descriptions, sometimes called *counterfactuals*, have a very different flavor from those we saw in the previous section.

Table 5-4. The NEX algorithm: Nonincremental induction of decision lists using exceptions.

```
Inputs: CSET is a set of two or more class names.
        ISET is a set of preclassified instances.
Output: A decision list composed of single-region descriptions.

Procedure NEX(CSET, ISET)

Let C be the most common class in ISET.
Initialize DLIST with C as the default class.
NEX-AUX(CSET, ISET, DLIST).
Return the decision list DLIST.

Procedure NEX-AUX(CSET, ISET, DLIST)

Let MISSED be the instances in ISET that DLIST misclassifies.
If MISSED is the empty set,
Then return DLIST.
Else let C be the most common class in MISSED.
     Induce a maximally specific description D that
        covers the instances of class C in MISSED.
     If D differs from the first term in DLIST,
     Then add 'If D, then C' to the beginning of DLIST.
          NEX-AUX(CSET, MISSED, DLIST).
     Else select a random attribute A with range [A1, A2].
          Let M1 be instances from MISSED with A < (A1-A2)/2.
          Let M2 be instances from MISSED with A ≥ (A1-A2)/2.
          NEX-AUX(CSET, M1, DLIST).
          NEX-AUX(CSET, M2, DLIST).
```

The NEX algorithm initializes its decision list by creating a default rule based on the most frequent class. On each iteration, the method applies its current decision list to the remaining training cases, to determine ones it misclassifies. NEX selects the most common class in this set, calls on a subroutine to induce the most specific description that covers the misclassified members of that class, and adds the resulting rule to the *front* of the decision list. The algorithm continues in this manner until the decision list correctly classifies all of the training instances.

For some unusual training sets, the algorithm may find the same description twice in succession, which would lead to an infinite recursion if allowed to continue. In such cases, NEX arbitrarily selects an attribute, computes the midpoint of its range, and divides the current set of training instances into two subsets, one having values less than the midpoint and another having values above it. The algorithm continues with this strategy until it breaks out of the problem and can return to its normal mode of operation.

Let us examine NEX's behavior on the numeric height/girth domain when only two classes are present. Figure 5-6 (a) shows the results when the algorithm operates using a conjunctive subroutine with a rectangular bias, such as the specific-to-general HSG method, which ignores negative instances. Since class + is most common, NEX has selected it as the default class. As an exception to this, the method has generated a rectangular region that just surrounds all instances of class −. However, since this area includes some + members, NEX has produced an exception to the exception, shown by the inner rectangle that just contains these + cases. Figure 5-6 (b) shows the analogous but somewhat different decision boundaries that emerge when NEX uses spherical threshold units to describe each region.

Clearly, given the same training instances and the same method for creating component terms, the NEX and NSC algorithms can induce very different decision lists. For some target concepts, one method will produce more accurate descriptions than the other and vice versa. Although variants on the NSC algorithm are much more widely reported in the literature, the field has yet to identify the conditions under which each method is desirable.

Typically, the decision region for each term in a decision list is determined by the description of that term. However, the terms generated by the NEX algorithm have just enough extent to cover the instances on which they are based. For a small training set, it seems likely that an enclosed term underestimates the extent of the true region. An alternative classification scheme assigns an instance to an enclosed term whenever the instance is closer to the surface of that term's region than to the surface of the enclosing term. This approach has a competitive flavor, with the overall decision boundaries being determined by pairs of terms.

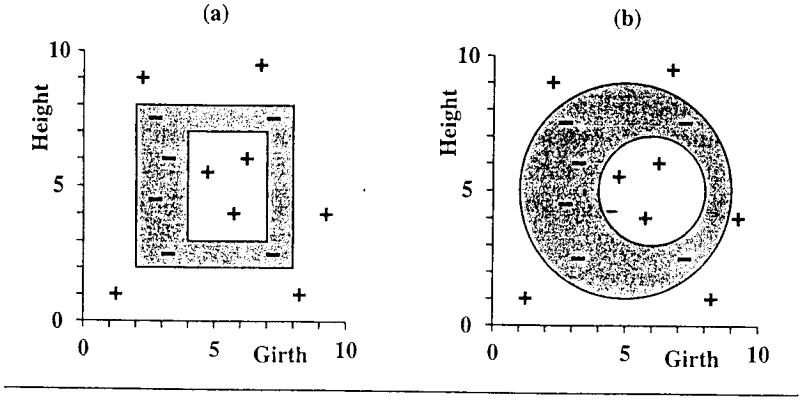

Figure 5-6. Decision boundaries produced by the NEX algorithm (a) when using the HSG routine to produce terms in the decision list and (b) when using a modified technique for spherical threshold units on the same data.

Incremental versions of the NEX algorithm are also possible. One variant, which we will call IEX (incremental induction using exceptions), stores no training cases. When the current decision list misclassifies an instance, this method checks the class associated with the term occurring just before the one that made the faulty prediction. If this has the desired class, then IEX generalizes the term just enough to let it cover the instance. If not, the algorithm adds a new term with the correct class, based on the training case, to the beginning of the decision list. The default rule uses the class of the initial training instance. Naturally, IEX loses information if it correctly classifies an instance for the wrong reason; storing the last k training cases reduces this effect, but any incremental hill-climbing scheme must pay some price of this sort.

5.5 Induction of competitive disjunctions

One can use an idea similar to those behind NSC and NEX to induce a disjunction of competitive descriptions. Table 5-5 presents an algorithm of this type, which we will call NCD (nonincremental induction of competitive disjunctions). This method invokes a simple competitive technique, like those we saw in Chapter 4, to create an initial set of descriptions. NCD then uses these summaries to classify the training set, noting which ones they assign correctly and which ones they misclassify.

Table 5-5. The NCD algorithm: Nonincremental induction of competitive disjunctions based on misclassified instances.

```
Inputs:  ISET is a set of classified training instances.
         CSET is a set of two or more class names.
Output:  A competitive disjunction (DISJUNCTS).

Procedure NCD(CSET, ISET)
For each class C in CSET,
    Average the instances of C to form a description.
    Add the resulting description to DISJUNCTS.
If DISJUNCTS correctly classifies all instances in ISET,
Then return DISJUNCTS.
Else let OLDSET be ISET.
     For each class C in CSET,
         For each class D other than C in CSET,
             Let MISSED be those instances of D in ISET that
                 DISJUNCTS classifies as members of C.
             Add class D' (which predicts D) to CSET.
             For each instance I in MISSED,
                 Relabel I as a member of class D'.
     If ISET equals OLDSET,
     Then return DISJUNCTS.
     Else return NCD(CSET, ISET).
```

At this point, the algorithm removes the problematic training cases from the original summaries and places them in separate 'pseudoclasses', which have a different internal name but predict the same class. NCD produces a new set of descriptions based on the new partition of the instances, again checks for misclassifications, and repeats the process until it arrives at a disjunction that correctly assigns all training cases to their classes or until it can make no further progress.[1]

The NCD algorithm relies on some method for competitive induction to construct its component terms. Let us examine the method's behavior when combined with a simple routine that averages training instances to form prototypes. Given the training data in Figure 5-7 (a),

1. This can occur when a small set of instances has such little effect on the decision boundaries that they remain misclassified by the new descriptions.

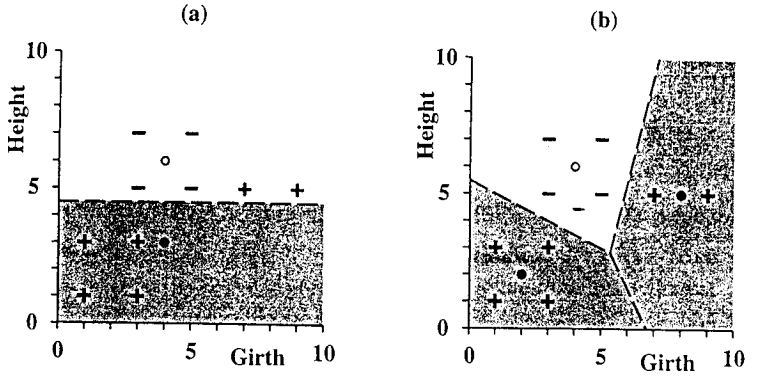

Figure 5-7. Prototypes (circles) and decision boundaries (dashed lines) produced (a) by a simple instance-averaging method and (b) by the NCD algorithm using an instance-averaging subroutine. The new decision regions cover the instances misclassified by the initial prototypes.

the algorithm computes a single prototype for each class, which gives the prototypes and decision boundaries shown. Because the resulting descriptions misclassify two of the class + instances, NCD assigns them to a new pseudoclass +′, and averages them to produce a +′ prototype. In addition, it removes these instances from the + average, giving the revised + prototype and decision boundaries in Figure 5-7 (b). This set of descriptions correctly classifies all training cases, so the process halts.

Clearly, one can combine the NCD algorithm with other competitive methods, such as a technique for constructing simple Bayesian classifiers, to obtain similar effects. But despite its elegance, one can easily design training sets that present difficulties for the NCD technique. Briefly, problems arise for the instance-averaging variant when the prototypes for two different classes lie near each other, and similar difficulties occur for the Bayesian version. This can produce misleading initial decision boundaries, which cause trouble on later iterations.

An alternative approach, which alleviates this problem, attempts to identify regions in another manner. One first groups instances into clusters that occur near each other (drawing on methods like those we discuss in Chapter 9), uses a competitive method like instance-averaging to generate descriptions for each class within a cluster, and then combines the results for each cluster into a single disjunction. The simplest

Table 5-6. The ICD algorithm: An incremental instance-averaging method.

```
Inputs: ISET is a set of classified training instances.
Output: A disjunction of single-region descriptions.

Procedure ICD(ISET).

Let the competitive description D be the empty set.
For each training instance I in ISET,
    Let C be the class name associated with I.
    If there are no terms that predict class C,
    Then add a new term to D based on I that predicts C.
    Else let T be the term in D that best matches I.
         If the term T predicts class C and if the
            match between I and T is good enough,
         Then update the term T by incorporating I.
         Else add a new term to D based on instance I.
Return the competitive description D.
```

method of this sort averages instances only if they are sufficiently close to each other, although this requires the user to specify a minimum distance. Because this scheme does not average instances from distant regions, it does not produce prototypes that fall in areas where no instances of the class occur.

One can embody similar ideas in incremental methods for inducing competitive disjunctions. Table 5-6 presents pseudocode for an algorithm of this sort, which we will call ICD (incremental induction of competitive disjunctions). Given a training instance, the technique finds the best matching of its disjunctive terms. If this description predicts the class of the instance, and if the match is good enough, then ICD averages the instance into the description for the term. Thus, it incorporates both the error-driven and clustering ideas we described above for nonincremental approaches. If these conditions are not met, then ICD adds a new term to its disjunction, basing the description on the attribute values and class of the training instance.

Figure 5-8 traces the behavior of an instance-averaging version of the ICD algorithm in the height/girth domain that involves two classes (+ and −). Here the test for a 'good enough' match means that the instance is no greater than a Euclidean distance of 5 from a prototype.

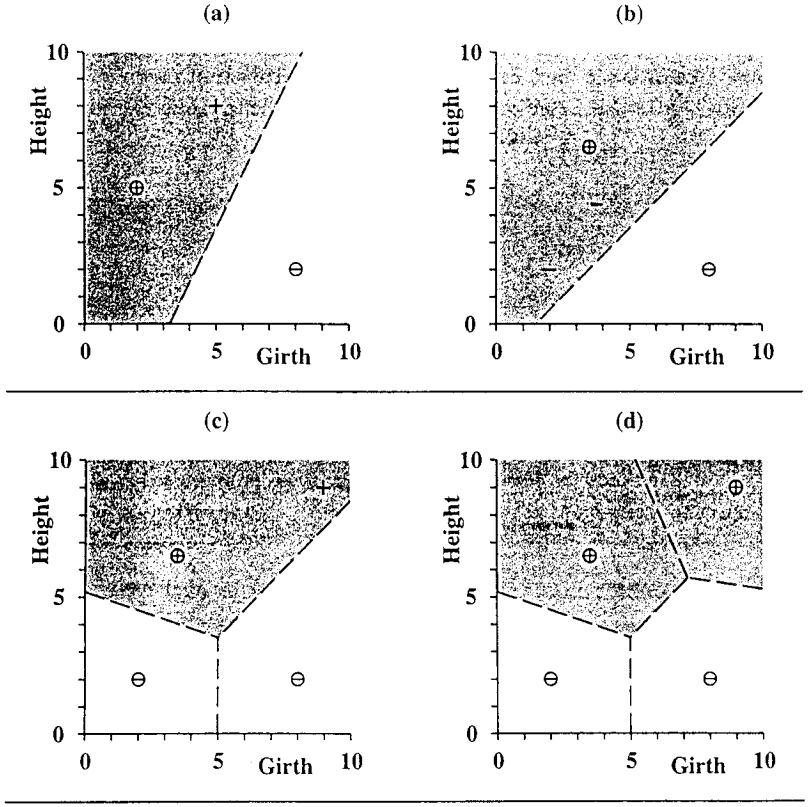

Figure 5-8. Behavioral trace of the ICD algorithm in response to three training instances, starting with one prototype for each class based on one instance each. The first instance is correctly classified (a), producing a revised prototype for +; the second instance is misclassified (b), giving a second - prototype; and the third instance is correctly labeled (c), but far enough away to produce a second + prototype (d).

After seeing one instance of class + and one of class -, the algorithm has one prototype for each class, as shown in Figure 5-8 (a). Given a new training instance of class + that falls within the decision region and the distance limit for this class's prototype, the learning method averages them, producing the revised prototype and decision boundaries in Figure 5-8 (b). Given a fourth instance, this time a member of the - class that falls within +'s decision region and thus is misclassified, the

ICD algorithm uses the instance to initialize a second prototype for class -, generating the revised decision boundaries in Figure 5-8 (c). Finally, given an instance that the + prototype correctly classifies but that is more than 5 units away, ICD creates a second prototype for this class.

Bayesian versions of the ICD algorithm are also possible, which incrementally produce disjunctions of probabilistic summaries, each with an associated base rate and a probability distribution for the attributes. If the class of training instance I agrees with the class predicted by the most probable description, and if this probability is high enough, then ICD incorporates the instance into the description, using it to update the base rate and probability distributions. However, if the instance differs sufficiently from all stored descriptions of the specified class, then the algorithm introduces a new description for that class based on the training instance.

One way to implement the 'good enough' parameter in a Bayesian framework is to use a *coupling probability*, c, which specifies the prior probability that any two instances in a class come from the same decision region. In this case, the prior probability that an instance belongs to a known region k is

$$\frac{cn_k}{(1-c)+cn} \quad ,$$

where n_k is the number of instances one has assigned to k and n is the total number of instances. On the other hand, the prior probability that an instance belongs to a new region is

$$\frac{(1-c)}{(1-c)+cn} \quad .$$

Inserting these terms into Bayes' rule (see Chapter 4) lets one determine the overall probability that a new instance belongs to a known region and to a novel one. In the former case, the 'good enough' condition is met and ICD averages the instance into the most probable description; in the latter, the instance leads to a new term in the competitive disjunction.

One drawback of the ICD method is its sensitivity to the 'good enough match' parameter. If the user sets this parameter too low, then the algorithm will store more prototypes than necessary and learning will be slow. On the other hand, if the user sets the parameter too high, then effects of training order come into play. For example, suppose ICD first encounters two instances of class A that belong in different regions

which are separated by a region of class B that it has not yet seen. In this case, the algorithm would decide that the two A instances belong to the same prototype and thus would average them. The resulting prototype could fall in the midst of the B area, and the technique would have no simple way to recover, other than eventually adding enough B prototypes near the offender to effectively guard against its retrieval, but this could take many training instances.

5.6 Instance-storing algorithms

Extending the clustering approach to its natural limit leads to the idea of storing a single prototype for each training instance. As with instance-averaging methods, one can use such knowledge to find the stored prototype that is closest to the new instance, retrieve its class, and predict this as the class of the new instance. This method, known as the *simple nearest neighbor* algorithm, produces the same results whether it operates incrementally, storing one case at a time, or nonincrementally, adding all training instances to memory en masse. Moreover, it makes no distinction between the class name and other attributes, and thus can be viewed as unsupervised.

Figure 5-9 (a) shows the decision boundaries generated by the simple nearest neighbor method on the training data from our previous example. Clearly, this approach partitions the instance space into more regions than the NCD algorithm or the clustering method. In fact, given n training instances, the simple instance-storing scheme can produce up to $n(n-1)/2$ decision boundaries. However, since some of these may be redundant, the effective number may be smaller; in the figure, only seven of the possible ten boundaries actually come into play. In any case, this approach produces different predictions on some test cases than does the instance-averaging approach.

As we noted in Section 5.1, allowing more than one description per class opens the door to combining evidence from them all when making a prediction. Given an instance-based representation, one can use a k *nearest neighbor* algorithm, which bases its decisions on a vote taken by the k nearest instances to the test case. Such voting techniques may reduce the sensitivity of instance-based techniques to irrelevant attributes, although one can also combine them with attribute-weighting methods like the one we saw in Chapter 4. Similar approaches are possible with

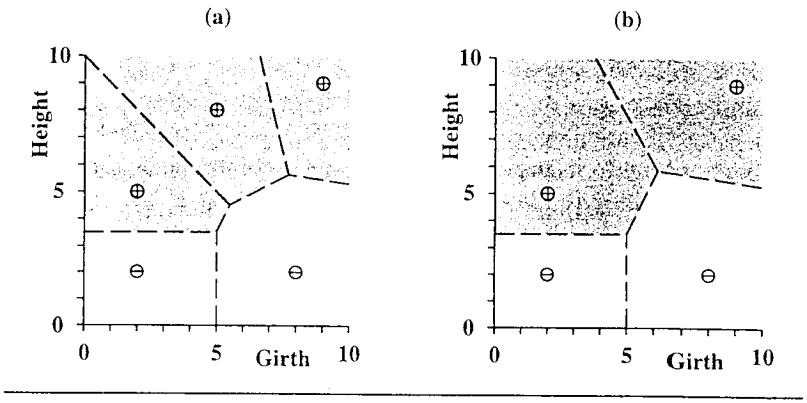

Figure 5-9. Decision boundaries produced for the training data from Figure 5-8 using (a) the simple nearest neighbor method, which stores all instances, and (b) the ECD algorithm, which stores instances only when it makes an error.

disjunctions of probabilistic summaries. These techniques produce a greater variety of decision boundaries than do simple nearest neighbor, with the details depending on the voting and matching schemes.

Simple instance-averaging and instance-storing methods fall at opposite ends of a spectrum, with the first storing one prototype per class and the latter storing every training case. The ICD algorithm provides a compromise position, storing more descriptions than the number of classes when necessary but fewer than the number of training cases. However, this solution approaches the problem from the instance-averaging perspective, and another compromise involves the selective storage of individual instances. Techniques of this sort are sometimes called *edited nearest neighbor* methods.

One such incremental method requires only two minor modifications of the ICD algorithm – removal of the averaging step and removal of the sufficient match condition. The resulting technique, which we will call ECD (error-driven induction of competitive disjunctions), learns only when it makes an error, storing a new prototype based on the misclassified training instance. In some cases, this scheme produces quite different behavior than selective instance averaging.[2]

2. No Bayesian versions of ECD exist, since some form of averaging is central to probabilistic methods.

For example, Figure 5-9 (b) shows the prototypes and decision boundaries that result when ECD is given the same training data, in the same order, as ICD in Figure 5-8. In this case, the second + prototype (in the upper right) has resulted not from exceeding a distance limit but from a classification error. Moreover, ECD has not modified the initial + prototype to incorporate the second + instance; since the former correctly classified the latter, it is simply forgotten. If the algorithm had observed the second + instance first, it would have stored this in memory and correctly classified both other + instances without needing to create another prototype. Thus, this method is also subject to order effects, although they seem relatively benign.

The ECD scheme tends to store prototypes that lie along the boundary of the target concept, as prediction errors tend to occur there. This observation has led to an interesting theoretical result for two-class domains. Let the target concept be a set of convex polygons having B as their total boundary length and A as their total area. Then the number of instances stored by the ECD method (and thus the number it must observe to reach a specified accuracy) is directly proportional to B/A. For a given area, more disjoint polygons mean greater boundary length, but the controlling factor is B/A, not the number of regions. In higher-dimensional spaces, this term generalizes to the ratio of the surface area and the volume, which grows rapidly with increasing dimensions.

5.7 Complementary beam search for disjunctive concepts

In Chapter 2 we encountered an interesting class of methods – genetic algorithms – that embodied a form of search that was quite distinct from others we considered. The basic method carried out a beam search that relied on genetic operators like mutation and crossover, rather than ones based on a generality ordering. Although we presented genetic algorithms initially as a technique for inducing logical conjunctions, they require only minor alterations to handle disjunctive domains, and they have interesting characteristics in this context as well.

5.7.1 Competitive and complementary descriptions

In Chapter 2 we saw that GA maintains a population of rules, each with an associated score on some evaluation metric. We viewed these rules as competitors, with each vying for precedence during classification,

and this made excellent sense when the target concept was conjunctive in nature. However, in disjunctive domains one can view the same population as a decision list, with scores indicating rules' position in the ordering. In this framework, different rules for the same class handle different (possibly disjoint) regions of the instance space. In terms of the evolutionary metaphor, they occupy different *niches* in the environment.

The only difficulty lies in the genetic algorithm itself, which tends to converge on a population that contains many copies of a single high-scoring description. For a disjunctive domain, we would like a modified method that converges toward a number of distinct descriptions or 'species', one for each region or 'niche' in the instance space. If the best description for one region has a higher score than that for another region, we would like the number of candidates in the population to vary in direct proportion to that score.

One way to obtain this effect is to use a *sharing function* in determining the probability of parenthood. The basic idea is that similar descriptions must share 'resources' in the environment, thus giving a bias toward diversity in the population. For example, suppose we let the similarity $B(x_i, x_j)$ between two descriptions x_i and x_j be the number of bits they share, and let $S(x_i)$ be the unmodified score for x_i. Then one can formulate a modified score as

$$S'(x_i) = \frac{S(x_i)}{\sum_{j=1}^{k} B(x_i, x_j)} \quad ,$$

where k is the number of features in each description. The expression produces lower scores for descriptions that are similar to many others, but has little effect on descriptions that are nearly unique. This scheme encourages niche formation, with the number of competitors within each niche gradually becoming proportional to the unmodified scores.

5.7.2 An example of GA inducing a decision list

To clarify the idea of niche formation, reconsider the training data from Figure 5-4, which we examined earlier in the context of the NSC method. Let us assume that the evaluation metric used to produce the unmodified score is $(P_c+N_{\bar{c}})/(P+N)$, which ranges from 0 to 1, and a `Beam-Size` or 'population size' of 8. Given an initial population of randomly generated logical conjunctions, and using the above sharing scheme to determine

Table 5-7. An eight-rule population that might be generated by the GA algorithm after a number of generations, given the training data in Figure 5-4, and fitness scores based on a simple accuracy measure and on a sharing function.

Vector	Logical conjunction	Initial	Shared
1 1 * *	two nuclei ∧ two tails	0.8	0.050
1 1 * *	two nuclei ∧ two tails	0.8	0.050
1 1 * *	two nuclei ∧ two tails	0.8	0.050
1 1 * *	two nuclei ∧ two tails	0.8	0.050
* * 1 1	thick wall ∧ dark color	0.7	0.050
* * 1 1	thick wall ∧ dark color	0.7	0.050
* 0 1 1	one tail ∧ thick wall ∧ dark color	0.6	0.043
* 0 1 1	one tail ∧ thick wall ∧ dark color	0.6	0.043

the probability of parenthood, the GA algorithm could, after sufficient iterations, produce a population like the one shown in Table 5-7.

There are a number of aspects of the table worth noting. First, only three descriptions exist in the population, one occurring four times and the others appearing twice each. Second, the unmodified score for the first description is 0.8, higher than the 0.7 and 0.6 results for the other two terms. This is because it covers four of the positive instances, whereas the others cover only three and two, respectively.

However, the modified score based on sharing is $\frac{0.8}{4\times4} = 0.05$ for the first conjunction, since it shares four bits with four members of the population (including itself) and none with the others. Similarly, the modified score for the second and third conjunctions is $\frac{0.7}{2\times4+2\times3} = 0.05$, since they share four bits with two descriptions, three with two others, and zero with the remaining five. The analogous computation for the third description is $\frac{0.6}{2\times4+2\times3} = 0.043$, based on the same demoninator as for the second.

The sharing scheme reduces the score for the first description because there are many similar candidates in competition. The same effect occurs for the second and third description, which compete with each other and with copies of themselves. However, in their case the reduction is less because there are fewer descriptions in the same niche. In this light, one can view the first description as being complementary to the other two, since it competes with them for 'resources' in only minor ways.

The long-term result of this 'evolutionary pressure' is a set of descriptions that appear in the population with a frequency proportional to their unmodified scores. In this case, the first description occurs $\frac{4}{8}$ of the time, whereas the other two, taken together, appear with the same probability. Each class of description has filled its 'niche' to capacity, producing a decision list that covers the positive instances but not the negatives.

5.7.3 Variations on complementary beam search

The above scenario presents only one use of genetic algorithms in the induction of logical disjunctions. One could use an alternative evaluation function to measure fitness (e.g., that takes simplicity into account), one could use a different similarity metric, and one could use a different population size. Nor are genetic methods necessarily limited to logical descriptions; the same basic scheme should work with simple threshold concepts or even competitive descriptions as members of the population. The extension to multi-class settings is straightforward; one simply includes additional features in the description language to represent the class attribute.

We should also note that the idea of complementary beam search is not limited to genetic algorithms. For example, one can use the same control regimen with induction operators that take advantage of the generality ordering on logical conjunctions. Nonincremental versions of this strategy always move from general to specific or vice versa, evaluating terms over the entire training set rather than over subsets, as in the NSC and NEX techniques. Incremental variants retain a set of weighted terms or rules in memory and employ bidirectional operators like those in ISC, creating more general candidates when they fail to match positive instances and producing more specific ones when they match incorrectly. Candidates that fare poorly are either dropped from memory or eventually transformed into useful descriptions, which complement one another by covering different decision regions, rather than competing to cover the same instances.

Naturally, both this approach and the genetic scheme are limited by the total number of descriptions retained in memory, so that if the target concept involves more decision regions than memory can hold, the learning algorithm will only find a partially correct description, even in

a noise-free environment. Thus, the user must make some assumption about the maximum number of regions in the domain before setting out, although this should not pose a problem for target concepts of moderate complexity.

5.8 Summary of the chapter

In this chapter we examined the induction of disjunctive descriptions. We found that such disjunctions organize their terms – simple logical, threshold, or competitive descriptions – into combinations that generate multiple decision regions; thus, they can represent a much wider range of concepts than the formalisms we encountered in previous chapters. However, the resulting space of disjunctive descriptions is accordingly more complex and requires new algorithms to search it, most of which use methods for inducing single-region logical, threshold, or competitive concepts as subroutines.

We saw that algorithms for disjunctive induction differ mainly in the conditions under which they invoke the simpler methods, and in these conditions' effect on the resulting organization of memory. For example, the nonincremental NSC uses a separate-and-conquer technique, finding a simple description D that covers some positive instances, adding D to its disjunctive hypothesis, removing the covered cases from consideration, and repeating the process until it has covered all positive instances. The NEX method uses a similar control structure, but it is driven by exceptions rather than uncovered instances, and adds new terms to the front of its decision list rather than to the end.

Both schemes can operate with either logical or threshold terms, but competitive representations require slightly different methods. Like NSC and NEX, the NCD algorithm is driven by classification errors and generates only enough decision regions to eliminate them, but it produces a competitive disjunction rather than an ordered decision list. In contrast, the nearest neighbor technique simply stores all training cases, creating one decision region for each observed instance.

Incremental versions exist for most of these methods. For example, ISC employs a strategy similar to NSC but invokes bidirectional operators that are driven by individual training cases. Similar relations hold between the NEX and NCD algorithms and their incremental variants, which can be sensitive to the order of training instances. The latter does not hold for the nearest neighbor scheme, which produces the same results when run incrementally or nonincrementally.

The above techniques carry out greedy or hill-climbing search through the space of disjunctive descriptions. Thus, their computational cost is tractable, provided this also holds for the routines they call to construct individual terms in the disjunction. Exhaustive search schemes might produce better results, but only at an exponential cost. Practical methods of the sort we have described exchange guarantees of optimality for efficiency during the learning process.

Most disjunctive methods place no restriction on the number of terms in each disjunction.[3] The NSC, NEX, and NCD algorithms introduce new terms only when they find training instances that are misclassified by existing ones, as do their incremental counterparts. Two exceptions to the above rule are the simple instance-storing algorithm, which stores exactly as many terms as training instances, and the genetic algorithm GA, which retains as many terms as allowed by its population size (although this specifies the maximum and not the effective number, since a term may occur multiple times).

Many variations exist on disjunctive induction, but the methods we have examined in this chapter constitute the main approaches found in the literature. Despite their simplicity, algorithms of this sort have found wide use and they have produced encouraging results in experimental studies of both natural and artificial induction tasks.

Exercises

1. Interchange the labels on positive and negative instances in Figure 5-2, then show the decision regions generated by the NSC algorithm (Section 5.2) when using the HSG subroutine on each pass. Do the same for variants of NSC that use HGS and a technique for learning spherical units, as in Figure 5-3.
2. Interchange the labels on positive and negative instances in Figure 5-4, then show the logical disjunction (using the cell notation) that NSC produces with the HGS subroutine to find individual rules. Show the same result when using the HSG algorithm instead.

3. Some theoretical results focus on k-DNF expressions, where each term contains at most k Boolean features. However, there are fewer results on k-term DNF expressions, in which each disjunction contains at most k terms.

3. Trace the hypotheses generated by the ISC algorithm (Section 5.3) on the training cases in Figure 5-4. 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. Use the same evaluation metric ($a + \frac{1}{t}$) as in the text.
4. Draw the decision regions generated by the NEX algorithm (Section 5.4) given the training cases in Figure 5-2, using HSG as a subroutine. Do the same using a technique for inducing spherical units. In both cases, treat + as the majority (default) class. Repeat the process with both subroutines using - as the majority class.
5. Remove the five outermost positive cases from Figure 5-6, leaving three positive instances and six negatives. Show the decision regions produced by a simple instance-averaging method and by a version of the NCD algorithm (Section 5.5) that invokes instance-averaging as a subroutine.
6. Assume that the positive instance in Figure 5-8 (a) is negative, but that the other cases retain the same labels. Trace the decision boundaries produced by the ICD algorithm after it has processed each of these training cases.
7. Show the decision regions produced by simple nearest neighbor and by the ECD algorithm (Section 5.6) on the data from Exercise 6.
8. Compute the fitness scores for the rules in Table 5-7 on the modified data from Exercise 2, using both the initial and sharing functions from Section 5.7. Select two of the highest-scoring rules on the latter metric and show two offspring that could result from a single application of the crossover operator (Chapter 2).

Historical and bibliographical remarks

Most work on the induction of DNF descriptions and decision lists has occurred within the framework of logical induction and has built on variants of the HSG and HGS algorithms (Chapter 2). Rivest (1987) defines the notion of decision lists and presents a formal analysis of their induction. Michalski and Chilausky's (1980) AQ system was one of the earliest examples of the nonincremental separate-and-conquer (NSC and MSC) methods. Clark and Niblett's (1989) CN2 and Michalski, Mozetic, Hong, and Lavrac's (1986) AQ15 constitute more recent instances of this approach, and Quinlan's (1990) FOIL and Muggleton and Feng's (1992) GOLEM extend the same scheme to relational domains.

A few researchers have adapted the separate-and-conquer technique to nonlogical frameworks, as in Marchand and Golea's (1993) construction of decision lists composed of linear threshold terms rather than conjunctive ones.

Papers on inducing decision lists through exceptions have been less frequent than those on separate-and-conquer methods, but Vere (1980) and Helmbold, Sloan, and Warmuth (1990) describe systems that use variants of the NEX algorithm. Work on incremental hill-climbing methods for decision-list induction have been even less common, although Iba et al.'s (1988) HILLARY incorporates a strategy identical to the ISC algorithm. Bradshaw (1987), Anderson and Matessa (1992), and a few others provide examples of the ICD approach to learning competitive disjunctions, but nonincremental variants like NCD are difficult to find in the literature.

Research on instance-storing methods like nearest neighbor has a long history in pattern recognition. Nilsson (1965) describes some early work along these lines, Dasarathy (1990) contains a broad sample of work from this perspective, and Cover and Hart (1967) report theoretical results about the asymptotic accuracy of such techniques. This approach to induction, first proposed by Fix and Hodges (1951), gained attention within machine learning only in the late 1980s, when Kibler and Aha (1987), Moore (1990), and others introduced them to the literature. More recent work along these lines has been reported by Aha, Kibler, and Albert (1991), Cardie (1993), and Moore and Lee (1994). Proceedings of annual workshops on case-based reasoning contain numerous papers on nearest neighbor and related techniques. Our brief treatment reflects not the large number of published papers on this topic, but rather the simple nature of most methods.

There also exists a large literature on genetic algorithms and their use in machine learning. Holland (1986) introduces one broad class of such methods, *classifier systems*, that use the complementary beam search approach in Section 5.7, while Booker, Goldberg, and Holland (1989) review early work in this area. Wilson (1987) and Booker (1988) present examples of the genetic induction of decision lists, and many more appear in proceedings of meetings on genetic algorithms published annually since 1985. Some work in adaptive production systems (Anderson & Kline, 1979; Langley, 1987) takes a similar approach but uses specialization or generalization operators rather than crossover or mutation.