LECTURE NOTES

ON

MACHINE LEARNING

VIII Semester **(**ACS014**)**

Mrs. G Sulakshana, Assistant Professor

Mr. A Praveen, Assistant Professor

Mrs. B Anupama, Assistant Professor

INFORMATION TECHNOLOGY

INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal – 500043, Hyderabad

UNIT-1

TYPES OF MACHINE LEARNING

Concept learning: Introduction, version spaces and the candidate elimination algorithm; learning with trees: Constructing decision trees, CART, classification example.

Alternatively, each concept can be thought of as a Boolean-valued function defined over this larger set (e.g., a function defined over all animals, whose value is true for birds and false for other animals). In this chapter we consider the problem of automatically inferring the general definition of some concept, given examples labeled as+.membersor nonmembers of the concept. This task is commonly referred to as *concept learning* or approx-imating a Boolean-valued function from examples.

A CONCEPT LEARNING TASK

To ground our discussion of concept learning, consider the example task of learn-ing the target concept "days on which my friend Aldo enjoys his favorite water sport." Table 2.1 describes a set of example days, each represented by a set of *attributes.* The attribute *Enjoy Sport* indicates whether or not Aldo enjoys his favorite water sport on this day. The task is to learn to predict the value of *Enjoy Sport* for an arbitrary day, based on the values of its other attributes. What hypothesis representation shall we provide to the learner in this case? Let us begin by considering a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. In particular, let each hypothesis be a vector of six constraints, specifying the values of the six attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast.* For each attribute, the hypothesis will either

- Indicate by a "?'that any value is acceptable for this attribute,
- specify a single required value (e.g., *Warm)* for the attribute, or
- Indicate by a *"0"* that no value is acceptable.

If some instance x satisfies all the constraints of hypothesis h, then h classifies x as a positive example (h(x) = 1). To illustrate, the hypothesis that Aldo enjoys his favorite sport only on cold days with high humidity (independent of the values of the other attributes) is represented by the expression.

Example	Sky	*Airtime*	*Humidity*	*Wind*	*Water*	*Forecast*	*Enjoy Sport*
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

TABLE 2.1

Positive and negative training examples for the target concept *Enjoy Sport.*

MACHINE LEARNING

Given:

- Instances *X:* Possible days, each described by the attributes
- *Sky* (with possible values *Sunny*, *Cloudy*, and *Rainy),*
- *AirTemp* (with values *Warm* and *Cold),*
- *Humidity* (with values *Normal* and *High),*
- *Wind* (with values *Strong* and *Weak),*

- *Water* (with values *Warm* and *Cool),* and

Forecast (with values *Same* and *Change).*

Hypotheses *H:* Each hypothesis is described by a conjunction of constraints on the attributes *Sky, AirTemp, Humidity, Wind, Water,* and *Forecast.* The constraints may be "?" (any value is acceptable), *" 0* (no value is acceptable), or a specific value. Target concept *c: EnjoySport* : *X* + (0 , l)

Training examples D: Positive and negative examples of the target function (see Table 2.1).

A hypothesis *h* in *H* such that *h (x) = c(x)* for all *x* in X.

When learning the target concept, the learner is presented a set of *training examples,* each consisting of an instance *x* from X, along with its target concept value *c (x)* (e.g., the training examples in Table 2.1). Instances for which *c (x)* = 1 are called *positive examples,* or members of the target concept. Instances for which *C (X)*= 0 are called *negative examples,* or nonmembers of the target concept. We will often write the ordered pair *(x ,c (x))* to describe the training example consisting of the instance x and its target concept value *c (x)* . We use the symbol to denote the set of available training examples.

Given a set of training examples of the target concept *c* , the problem faced by the learner is to hypothesize, or estimate, *c* . We use the symbol H to denote the set of all possible hypotheses that the learner may consider regarding the identity of the target concept. Usually H is determined by the human designer's choice of hypothesis representation. In general, each hypothesis *h* in H represents a boolean-valued function defined over X; that is, *h* : X --+ {O, 1). The goal of the learner is to find a hypothesis *h* such that *h (x) = c (x)* for n X.

The Inductive Learning Hypothesis

Notice that although the learning task is to determine a hypothesis h identical to the target concept *c* over the entire set of instances X, the only information available about *c* is its value over the training examples. Therefore, inductive learning algorithms can at best guarantee that the output hypothesis fits the target concept over the training data. Lacking any further information, our assumption is that the best hypothesis regarding unseen instances is the hypothesis that best fits the observed training data. This is the fundamental Assumption of inductive learning, and we will have much more to say about it throughout this book. We state it here informally and will revisit and analyze this assumption more formally and more quantitatively in Chapters 5, 6,and 7. The inductive learning hypothesis. Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

CONCEPT LEARNING AS SEARCH

Concept learning can be viewed as the task of searching through a large space of hypotheses implicitly defined by the hypothesis representation. The goal of this search is to find the hypothesis that best fits the training examples. It is important to note that by selecting a hypothesis representation, the designer of the learning algorithm implicitly defines the space of all hypotheses that the program can ever represent and therefore can ever learn. Consider, for example, the instances X and hypotheses H in the *EnjoySport* learning task. Given that the attribute *Sky* has three possible values, and that *AirTemp, Humidity, Wind, Water,* and *Forecast* each have two possible values, the instance space X contains exactly 3. 2 2 . 2 2 . 2 = 96 distinct instances. A similar calculation shows that there are 5 . 4 - 4- 4- 4 . 4= 5 120 syntactically

distinct hypotheses within H. Notice, however, that every hypothesis containing one or more "IZI" symbols represents the empty set of instances; that is, it classifies every instance as negative. Therefore, the number of semantically distinct hypotheses is only 1+ (4 . 3 . 3 . 3 . 3 . 3)= 973. Our Enjoy Sport example is a very simple learning task, with a relatively small, finite hypothesis space. Most practical learning tasks involve much larger, sometimes infinite, hypothesis spaces.

If we view learning as a search problem, then it is natural that our study of learning algorithms will e x a ~ t h different strategies for searching the hypothesis space. We will be particularly interested in algorithms capable of efficiently searching very large or infinite hypothesis spaces, to find the hypotheses that best fit the training data.

General-to-Specific Ordering of Hypotheses

Many algorithms for concept learning organize the search through the hypothesis space by relying on a very useful structure that exists for any concept learning problem: a general-to-specific ordering of hypotheses. By taking advantage of this naturally occurring structure over the hypothesis space, we can design learning algorithms that exhaustively search even infinite hypothesis spaces without explicitly enumerating every hypothesis. To illustrate the general-to-specific ordering, consider the two hypotheses

hi = (Sunny, ?, ?, Strong, ?, ?)
h2 = (Sunny, ?, ?, ?, ?, ?)

Now consider the sets of instances that are classified positive by hl and by h2. Because h2 imposes fewer constraints on the instance, it classifies more

instances as positive. In fact, any instance classified positive by hl will also be classified positive by h2. Therefore, we say that h2 is more general than hl.

This intuitive "more general than" relationship between hypotheses can be defined more precisely as follows. First, for any instance x in X and hypothesis h in H, we say that x satisfies h if and only if h(x) = 1. We now define the *more - general Han _ or. - equal~o*relation in terms of the sets of instances that satisfy the two hypotheses: Given hypotheses hj and hk, hj is more-general-than-- equal do hk if and only if any instance that satisfies hk also satisfies *hi.*

Definition: Let *hj* and *hk be* Boolean-valued functions defined over X. Then *hj* is more general-than-or-equal-to *hk* (written *hj 2, h k)*if and only if

We will also find it useful to consider cases where one hypothesis is strictly more general than the other. Therefore, we will say that hj is (strictly) more-general

VERSION SPACES AND THE CANDIDATE-ELIMINATION ALGORITHM

This section describes a second approach to concept learning, the CANDIDATE-ELIMINATION algorithm that addresses several of the limitations of FIND-S. Notice that although FIND-S outputs a hypothesis from H, that is consistent with the training examples, this is just one of many hypotheses from H that might fit the training data equally well. The key idea in the CANDIDATE-ELIMINATION algorithm is to output a description of

the set of all hypotheses consistent with the train-ing examples. Surprisingly, the CANDIDATE-ELIMINATION algorithm computes the description of this set without explicitly enumerating all of its members. This is accomplished by again using the more-general-than partial ordering, this time to maintain a compact representation of the set of consistent hypotheses and to incrementally refine this representation as each new training example is encountered.

The CANDIDATE-ELIMINATION algorithm has been applied to problems such as learning regularities in chemical mass spectroscopy (Mitchell 1979) and learning control rules for heuristic search (Mitchell et al. 1983). Nevertheless, practical applications of the CANDIDATE-ELIMINATION and FIND-Algorithms are limited by the fact that they both perform poorly when given noisy training data. More importantly for our purposes here, the CANDIDATE-ELIMINATION algorithm provides a useful conceptual framework for introducing several fundamental is-sues in machine learning. In the remainder of this chapter we present the algorithm and discuss these issues. Beginning with the next chapter, we will ex-amine learning algorithms that are used more frequently with noisy training data.

Representation

The CANDIDATE-ELIMINATION algorithm finds all describable hypotheses that are consistent with the observed training examples. In order to define this algorithm precisely, we begin with a few basic definitions. First, let us say that a hypothesis is *consistent* with the training examples if it correctly classifies these examples.

Definition: A hypothesis h is consistent with a set of training examples *D* if and only if $h(x) = c(x)$ for each example (x, $c(x)$)in *D.*

Notice the key difference between this definition of *consistent* and our earlier definition of *satisfies.* An example *x* is said to *satisfy* hypothesis *h* when *h(x)* = 1, regardless of whether x is a positive or negative example of the target concept. However, whether such an example is *consistent* with *h* depends on the target concept, and in particular, whether *h (x) = c (x)* .The CANDIDATE-ELIMINATION algorithm represents the set of *all* hypotheses consistent with the observed training examples. This subset of all
Concept.

THE LIST-THEN-ELIMINATE ALGORITHM

One obvious way to represent the version space is simply to list all of its members.

This leads to a simple learning algorithm, which we might call the LIST-THEN-ELIMINATE algorithm, The LIST-THEN-ELIMINATE algorithm first initializes the version space to contain all hypotheses in H, then eliminates any hypothesis found inconsistent with any training example. The version space of candidate hypotheses thus shrinks as more examples are observed, until ideally just one hypothesis remains that are consistent with all the observed examples. This, presumably, is the desired target concept. If insufficient data is available to narrow the version space to a single

hypothesis, then the algorithm can output the entire set of hypotheses consistent with the observed data.

In principle, the LIST-THEN-ELIMINATE algorithm can be applied whenever the hypothesis space H is finite. It has many advantages, including the fact that it is guaranteed to output all hypotheses consistent with the training data. Unfortunately, it requires exhaustively enumerating all

hypotheses in H-an unrealistic requirement for all but the most trivial hypothesis spaces.

A More Compact Representation for Version Spaces

The CANDIDATE-ELIMINATION algorithm works on the same principle as the above LIST-THEN-ELIMINATE algorithm. However, it employs a much more compact representation of the version space. In particular, the version space is represented by its most general and least general members. These members form general and specific boundary sets that delimit the version space within the partially ordered hypothesis space.

THE LIST-THEN-ELIMINATE ALGORITHM

Version Space c a list containing every hypothesis in H

For each training example, *(x , c (x))* remove from *Version Space* any hypothesis *h* for which *h(x) # c (x)*

Output the list of hypotheses in *Version Space*

A version space with its general and specific boundary sets. The version space includes all six hypotheses shown here, but can be represented more simply by S and *G* . Arrows indicate instances of the *more-general-than* relation. This is the version space for the *Enjoyspor t* concept learning problem and training examples described in Table 2.1.

To illustrate this representation for version spaces, consider again the En-*joy sport* concept learning problem. Recall that given the four training examples from Table 2.1, FIND-S outputs the hypothesis

= *(Sunny, Warm,* ?, *Strong,* ?, ?)

In fact, this is just one of six different hypotheses from H that are consistent with these training examples. They constitute the version space relative to this

set of data and this hypothesis representation. The CANDIDATE-ELIMINATION algorithm rep-resents the version space by storing only its most general members and its most specific (labeled *S* in the figure). Given only these two sets *S* and G, it is possible to enumerate all members of the version space as needed by generating the hypotheses that lie between these two sets in the general-to-specific partial ordering over hypotheses.

It is intuitively plausible that we can represent the version space in terms of its most specific and most general members. Below we define the boundary sets and *S* precisely and prove that these sets do in fact represent the version space.

Definition: The general boundary G, with respect to hypothesis space *H* and training data *D*, is the set of maximally general members of *H* consistent with *D.*

G = {g E *HIConsistent(g, D) A (-3gf* E *H) [(g f>,g)* A *Consistent(gt,D)]]*

Definition: The specific boundary S, with respect to hypothesis space *H* and training data *D*, is the set of minimally general (i.e., maximally specific) members of *H* consistent with *D.*

S rn *{s* E *H(Consistent(s,D)* A *(-3s'* E *H)[(s >,s f)*A *Consistent(st,D)])*

As long as the sets *G* and *S* are well defined they completely specify the version space. In particular, we can show that the version space is precisely the set of hypotheses contained in *G*, plus those contained in *S*, plus those that lie between *G* and *S* in the partially ordered hypothesis space. This is stated precisely in Theorem 2.1.

Theorem 2.1. Version space representation theorem. Let X be an arbitrary set of instances and let H be a set of boolean-valued hypotheses defined over X. Let c : *X* + {O,1) be an arbitrary target concept defined over X, and let D be an arbitrary set of training examples {(x,c(x))). For all X, H, c, and D such that *S* and *G* are well defined,

Proof. To prove the theorem it suffices to show that (1) every h satisfying the right-hand side of the above expression is in *V S H, and*~ (2) every member of *V S H* , ~ satisfies the right-hand side of the expression. To show (1) let g be an arbitrary member of *G* , s be an arbitrary member of *S,* and h be an arbitrary member of H, such that g 2, h 2, s. Then by the definition of *S,* s must be satisfied by all positive examples in D. Because h 2, s, h must also be satisfied by all positive examples in

Similarly, by the definition of *G* , g cannot be satisfied by any negative example in D, and because g 2, h, h cannot be satisfied by any negative example in D. Because h is satisfied by all positive examples in D and by no negative examples in D, h is consistent with D, and therefore h is a member of *V S H* , ~This. Proves step (1). The argument for (2) is a bit more complex.

CANDIDATE-ELIMINATION LEARNING ALGORITHM

The CANDIDATE-ELIMINATION algorithm computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples. It begins by initializing the version space to the set of all hypotheses in H; that is, by initializing the *G* boundary set to contain the most general hypothesis in H

Go + {(?, ?, ?, ?, ?, ?)} and initializing the *S* boundary set to contain the most specific (least general) hypothesis

so c- ((@,PI,@,PI, 0,0)1

These two boundary sets delimit the entire hypothesis space, because every other hypothesis in H is both more general than *So* and more specific than *G o*

As each training example is considered, the *S* and *G* boundary sets are generalized and specialized, respectively, to eliminate from the version space any hypotheses found inconsistent with the new training example. After all examples have been processed, the computed version space contains all the

hypotheses consistent with these examples and only these hypotheses.

Initialize G to the set of maximally general hypotheses in H

Initialize S to the set of maximally specific hypotheses in H

For each training example d, do

- Add to S all minimal generalizations h of s such that
- h is consistent with d, and some member of G is more general than h
- Remove from S any hypothesis that is more general than another hypothesis in S

If d is a negative example

Remove from S any hypothesis inconsistent with d

Add to G all minimal specializations h of g such that h is consistent with d, and some member of *S* is more specific than h

- Remove from G any hypothesis that is less general than another hypothesis in G.

CANDIDATE-ELIMINATION algorithm using version spaces. Notice the duality in how positive and negative examples influence *S* and G.

Notice that the algorithm is specified in terms of operations such as computing minimal generalizations and specializations of given hypotheses, and identifying nonminimal and nonmaximal hypotheses. The detailed implementation of these operations will depend, of course, on the specific representations for instances and hypotheses. However, the algorithm itself can be applied to any concept learn-ing task and hypothesis space for which these operations are well-defined. In the following example trace of this algorithm, we see how such operations can be implemented for the representations used in the *EnjoySport* example problem.

An Illustrative Example

CANDIDATE-ELIMINATION algorithm applied to the first two training examples from Table 2.1. As described above, the boundary sets are first initialized to Go and So, the most general and most specific hypotheses in H, respectively.

When the first training example is presented (a positive example in this case), the CANDIDATE-ELIMINATION algorithm checks the *S* boundary and finds that it is overly specific-it fails to cover the positive example. The boundary is therefore revised by moving it to the least more general hypothesis that covers this new example. When the second training example (also positive) is observed, it has a similar effect of generalizing *S* further to *S2*, leaving G again unchanged (i.e., G2 = G I = GO). Notice the processing of these first

MACHINE LEARNING

S 1 : 1*{<Sunny,Warm,Normal,Strong,Warm,Same>}*

S2 : *{<Sunny, Warm,* ?, *Strong, Warm, Same>}*

Training examples:

1. <Sunny, Warm, Normal, Strong, Warm, Same>, Enjoy Sport = Yes

2. <Sunny, Warm, High, Strong, Warm, Same>, Enjoy Sport = Yes

CANDIDATE-ELIMINATION Trace1. *So* and *Go* are the initial boundary sets corresponding to the most specific and most general hypotheses. Training examples 1 and 2 force the *S* boundary to become more general, as in the FIND-S algorithm. They have no effect on the *G* boundary two positive

examples is very similar to the processing performed by the FIND-S algorithm.

As illustrated by these first two steps, positive training examples may force the S boundary of the version space to become increasingly general. Negative training examples play the complimentary role of forcing the G boundary to become increasingly specific. Consider the third training example, this negative example reveals that the G boundary of the version space is overly general; that is, the hypothesis in G incorrectly predicts that this new example is a positive example. The hypothesis in the G boundary must therefore be specialized until it correctly classifies this new negative example. As shown in Figure *2.5,* there are several alternative minimally more specific hypotheses.. All of these become members of the new G3 boundary set.

Given that there are six attributes that could be specified to specialize G2, why are there only three new hypotheses in G3? For example, the hypothesis = (?, ?, *Normal,* ?, ?, ?) is a minimal specialization of G2 that correctly labels the new example as a negative example, but it is not included in Gg. The reason this hypothesis is excluded is that it is inconsistent with the previously encountered positive examples. The algorithm determines this simply by noting that *h* is not more general than the current specific boundary, Sz. In fact, the S boundary of the version space forms a summary of the previously encountered positive examples that can be used to determine whether any given hypothesis

CANDIDATE-ELMNATION Trace 2. Training example 3 is a negative example that forces the G2 boundary to be specialized to G3.Note several alternative maximally general hypotheses are included in Gj is consistent with these examples. Any hypothesis more general than *S* will, by definition, cover

any example that *S* covers and thus will cover any past positive example. In a dual fashion, the G boundary summarizes the information from previously encountered negative examples. Any hypothesis more specific than G is assured to be consistent with past negative examples. This is true because any such hypothesis, by definition, cannot cover examples that G does not cover. The fourth training example, as shown in Figure 2.6, further generalizes the S boundary of the version space. It also results in removing one member of the G boundary, because this member fails to cover the new positive example. This last action results from the first step under the condition "If d is a positive example". To understand the rationale for this step, it is useful to consider why the offending hypothesis must be removed from G. Notice it cannot be specialized, because specializing it would not make it cover the new example. It also cannot be generalized, because by the definition of G, any more general hypothesis will cover at least one negative training example. Therefore, the hypothesis must be dropped from the G boundary, thereby removing an entire branch of the partial ordering from the version space of hypotheses remaining under consideration.

After processing these four examples, the boundary sets S4 and G4 delimit the version space of all hypotheses consistent with the set of incrementally ob-served training examples. The entire version space, including those hypotheses.

CANDIDATE-ELIMINATION Trace3. The positive training example generalizes the *S* boundary, from *S3* to *S4*. One member of *Gg* must also be deleted, because it is no longer more general than the *S4* boundary.

This learned version space is independent of the sequence in which the training examples are presented (be-cause in the end it contains all hypotheses consistent with the set of examples). As further training data is encountered,

the S and G boundaries will move monotonically closer to each other, delimiting a smaller and smaller version space of candidate hypotheses.

REMARKS ON VERSION SPACESAND CANDIDATE-ELIMINATI

Will the CANDIDATE-ELIMINATION Algorithm Converge to the Correct Hypothesis?

The version space learned by the CANDIDATE-ELIMINATION algorithm will con-verge toward the hypothesis that correctly describes the target concept, provided there are no errors in the training examples, and (2) there is some hypothesis in H that correctly describes the target concept. In fact, as new training examples are observed, the version space can be monitored to determine the remaining am-biguity regarding the true target concept and to determine when sufficient training examples have been observed to unambiguously identify the target concept. The target concept is exactly learned when the S and *G* boundary sets converge to a single, identical, hypothesis.

What will happen if the training data contains errors? Suppose, for example, that the second training example above is incorrectly presented as a negative example instead of a positive example. Unfortunately, in this case the algorithm is certain to remove the correct target concept from the version space! Because, it will remove every hypothesis that is inconsistent with each training example, it will eliminate the true target concept from the version space as soon as this false negative example is encountered. Of course, given sufficient additional training data the learner will eventually detect an inconsistency by noticing that the S and G boundary sets eventually converge to an empty version space. Such an empty version space indicates that there is no hypothesis in H consistent with all observed training examples. A similar

symptom will appear when the training examples are correct, but the target concept cannot be described in the hypothesis representation (e.g., if the target concept is a disjunction of feature attributes and the hypothesis space supports only conjunctive descriptions). We will consider such eventualities in greater detail later. For now, we consider only the case in which the training examples are correct and the true target concept is present in the hypothesis space.

What Training Example Should the Learner Request Next?

Up to this point we have assumed that training examples are provided to the learner by some external teacher. Suppose instead that the learner is allowed to conduct experiments in which it chooses the next instance, then obtains the correct classification for this instance from an external oracle (e.g., nature or a teacher). This scenario covers situations in which the learner may conduct experiments in nature (e.g., build new bridges and allow nature to classify them as stable or unstable), or in which a teacher is available to provide the correct classification (e.g., propose a new bridge and allow the teacher to suggest whether or not it will be stable). We use the term *query* to refer to such instances constructed by the learner, which are then classified by an external oracle. Consider again the version space learned from the four training examples of the *Enjoysport* concept and illustrated in Figure 2.3. What would be a good query for the learner to pose at this point?

DECISION TREE LEARNING

Decision tree learning is one of the most widely used and practical methods for inductive inference. It is a method for approximating discrete-valued functions that is robust to noisy data and capable of learning disjunctive expressions. This chapter describes a family of decision tree learning algorithms that includes widely used algorithms such as ID3, ASSISTANT, and C4.5. These decision tree learning meth-ods search a completely

expressive hypothesis space and thus avoid the difficulties of restricted hypothesis spaces. Their inductive bias is a preference for small trees over large trees.

INTRODUCTION

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be re-represented as sets of if-then rules to improve human readability. These learning methods are among the most popular of inductive inference algorithms and have been successfully applied to a broad range of tasks from learning to diagnose medical cases to learning to assess credit risk of loan applicants.

DECISION TREE REPRESENTATION

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the sub tree rooted at the new node. This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis. For example, the instance

(Outlook = Sunny, Temperature = Hot, Humidity = High, Wind = Strong)

Would be sorted down the left most branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that

PlayTennis = no). This tree and the example used in Table *3.2* to illustrate the ID3 learning algorithm are adapted from (Quinlan *1986).*

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests and the tree itself to a disjunction of these conjunctions.

APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

Although a variety of decision tree learning methods have been developed with somewhat differing capabilities and requirements, decision tree learning is generally best suited to problems with the following characteristics:

- Znstances are represented by attribute-value pairs. Instances are described by a fixed set of attributes (e.g., *Temperature)* and their values (e.g., *Hot).* The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., *Hot, Mild, Cold).* However, extensions to the basic algorithm allow handling real-valued attributes as well (e.g., representing *Temperature* numerically).
- The target function has discrete output values. a Boolean classification (e.g., *yes* or *no)* to each example. Decision tree methods easily extend to learning functions with more than two possible output values. A more substantial extension allows learning target functions with real-valued outputs, though the application of decision trees in this setting is less common. Disjunctive descriptions may be required. As noted above, decision trees naturally represent disjunctive expressions.

The training data may contain errors. Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.

The training data may contain missing attribute values. Decision tree methods can be used even when some training examples have unknown values (e.g., if the *Humidity* of the day is known for only some of the training examples).
Many practical problems have been found to fit these characteristics. Decision tree learning has therefore been applied to problems such as learning to classify medical patients by their disease, equipment malfunctions by their cause, and loan applicants by their likelihood of defaulting on payments. Such problems, in which the task is to classify examples into one of a discrete set of possible categories, are often referred to as *classification problems.*

THE BASIC DECISION TREE LEARNING ALGORITHM

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees. This approach is exemplified by the ID3 algorithm (Quinlan 1986) and its successor C4.5 (Quinlan 1993), which form the primary focus of our discussion here. In this section we present the basic algorithm for decision tree learning, corresponding approximately to the ID3 algorithm. we consider a number of extensions to this basic algorithm, including extensions incorporated into C4.5 and other more recent algorithms for decision tree learning.

Our basic algorithm, ID3, learns decision trees by constructing them top-down, beginning with the question "which attribute should be tested at the root of the tree?'To answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples. The best attribute is selected and used as the test at the root node of the tree. A descendant of the root node is then created for each possible value

of this attribute, and the training examples are sorted to the appropriate descendant node (i.e., down the branch corresponding to the example's value for this attribute). The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree. This forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices. A simplified version of the algorithm, specialized to learning Boolean-valued functions (i.e., concept learning).

Which Attribute Is the Best Classifier?

The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree. We would like to select the attribute that is most useful for classifying examples. What is a good quantitative measure of the worth of an attribute? We will define a statistical property, called *information gain that* measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select among the candidate attributes at each step while growing the tree.

ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

In order to define information gain precisely, we begin by defining a measure commonly used in information theory, called *entropy*, that characterizes the (im)purity of an arbitrary collection of examples. Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this Boolean classification is *ID3(Examples, Target attribute*, Attributes)

Examples are the training examples. Target attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that

correctly classifies the given Examples.

Summary of the *ID3* algorithm specialized to learning Boolean-valued functions. *ID3* is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples, or until all attributes have been used where p, is the proportion of positive examples in S and p, is the proportion of negative examples in S. In all calculations involving entropy we define 0 log 0 to be 0.

To illustrate, suppose S is a collection of 14 examples of some Boolean concept, including 9 positive and 5 negative examples (we adopt the notation [9+, 5-1 to summarize such a sample of data). Then the entropy of S relative to this Boolean classification is notice that the entropy is 0 if all members of S belong to the same class. For example, if all members are positive (pe = I), then p, is 0, and Entropy(S) = -1 . log2(1) - 0 . log2 0 = -1 . 0 - 0 . log2 0 = 0. Note the entropy is 1 when the collection contains an equal number of positive and negative examples. If the collection contains unequal numbers of positive and negative examples, the entropy is between 0 and 1. The form of the entropy function relative to a Boolean classification, as p, varies between 0 and 1.

One interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the classification of an arbitrary member of *S* (i.e., a member of *S* drawn at random with uniform probability). For example, if *p* , is 1, the receiver knows the drawn example will be positive, so no message need be sent, and the entropy is zero. On the other hand, if *pe* is 0.5, one bit is required to indicate whether the drawn example is positive or negative. If *pe* is 0.8, then a collection of messages can

be encoded using on average less than 1 bit per message by assigning shorter codes to collections of positive examples and longer codes to less likely negative examples.

Thus far we have discussed entropy in the special case where the target classification is Boolean. More generally, if the target attribute can take on c

different values, then the entropy of *S* relative to this c-wise classification is defined as Where *pi* is the proportion of *S* belonging to class *i.* Note the logarithm is still base 2 because entropy is a measure of the expected encoding length measured in *bits.* Note also that if the target attribute can take on c possible values, the entropy can be as large as log, *c.*

INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called *information gain,* is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, *Gain(S, A)* of *an* attribute A, relative to a collection of examples *S,* is defined where *Values(A)* is the set of all possible values for attribute A, and *S,* is the subset of S for which attribute A has value v (i.e., *S, = { s E SIA(s) = v))* . Note the first term in Equation *(3.4)* is just the entropy of the original collection *S,* and the second term is the expected value of the entropy after *S* is partitioned using attribute *A.* The expected entropy described by this second term is simply the sum of the entropies of each subset *S,* weighted by the fraction of examples that belong to *S,. Gain(S,A)* is therefore the expected reduction in

entropy caused by knowing the value of attribute A. Put another way, *Gain(S,A)* is the information provided about the *target &action value*, given the value of some other attribute *A.* The value of *Gain(S, A)* is the number of bits saved when encoding the target value of an arbitrary member of *S,* by knowing the value of attribute *A.*

For example, suppose *S* is a collection of training-example days described by attributes including *Wind,* which can have the values *Weak* or *Strong.* As before, assume *S* is a collection containing *14* examples, [9+, 5-1. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have *Wind = Weak,* and the remainder have *Wind = Strong.* The information gain due to sorting the original *14* examples by the attribute *Wind* may then be calculated as

Values (Wind) = Weak, Strong

Information gain is precisely the measure used by ID3 to select the best attribute at each step in growing the tree. The use of information gain to evaluate the relevance of attributes is summarized. In this figure the information gain of two different attributes, *Humidity* and *Wind,* is computed in order to determine which is the better attribute for classifying the training examples.

Humidity provides greater information gain than *Wind,* relative to the target classification. Here, E stands for entropy and *S* for the original collection of examples. Given an initial collection *S* of 9 positive and 5 negative examples, [9+, 5-1, sorting these by their *Humidity* produces collections of [3+, 4-1 *(Humidity = High)* and [6+, 1-1 *(Humidity = Normal).* The information gained by this partitioning is .151, compared to a gain of only .048for the attribute *Wind.*

An Illustrative Example

To illustrate the operation of ID3, consider the learning task represented by the training examples. Here the target attribute *Play Tennis*, which can have values *yes* or *no* for different Saturday mornings, is to be predicted based on other attributes of the morning in question. Consider the first step through

Day	*Outlook*	*Temperature*	*Humidity*	*Wind*	*Play Tennis*
D l	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
Dl0	Rain	Mild	Normal	Weak	Yes
Dl1	Sunny	Mild	Normal	Strong	Yes
Dl2	Overcast	Mild	High	Strong	Yes
Dl3	Overcast	Hot	Normal	Weak	Yes
Dl4	Rain	Mild	High	Strong	No

TABLE 3.2

Training examples for the target concept *PlayTennis*.the algorithm, in which the topmost node of the decision tree is created. Which attribute should be tested first in the tree? ID3 determines the information gain for each candidate

attribute (i.e., *Outlook, Temperature, Humidity,* and *Wind),* then selects the one with highest information gain. The computation of information gain for two of these attributes. The information gain values for all four attributes are

- *Gain(S, Outlook)* = 0.246
- *Gain(S,Humidity)* = 0.151
- *Gain(S,Wind)* = 0.048
- *Gain(S, Temperature)* = 0.029

Where *S* denotes the collection of training examples from Table 3.2.

According to the information gain measure, the *Outlook* attribute provides the best prediction of the target attribute, *PlayTennis,* over the training examples. Therefore, *Outlook* is selected as the decision attribute for the root node, and branches are created below the root for each of its possible values (i.e., *Sunny, Overcast,* and *Rain).* The resulting partial decision tree in along with the training examples sorted to each new descendant node. Note that every example for which *Outlook* = *Overcast* is also a positive ex-ample of Play Tennis. Therefore, this node of the tree becomes a leaf node with the classification *PlayTennis* = *Yes.* In contrast, the descendants corresponding to *Outlook* = *Sunny* and *Outlook* = *Rain* still have nonzero entropy, and the decision tree will be further elaborated below these nodes.

The process of selecting a new attribute and partitioning the training examples is now repeated for each non tenninal descendant node, this time using only the training examples associated with that node. Attributes that have been incorporated higher in the tree are excluded, so that any given attribute can appear at most once along any path through the tree. This process continues for each new leaf node until either of two conditions is met: (1) every attribute has already been included along this path through the tree, or (2) the training examples associated with this leaf node all have the same target attribute value

(i.e., their entropy is zero).

HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

As with other inductive learning methods, ID3 can be characterized as searching a space of hypotheses for one that fits the training examples. The hypothesis space searched by ID3 is the set of possible decision trees. ID3 performs a simple-to-complex; hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data. The evaluation function

{Dl, D2, ..., Dl41

P+S-I

Which attribute should be tested here?

- *Gain (Ssunnyj Temperaare)* = ,970 - (215) 0.0 - (Y5) 1.0 - (115) 0.0 = ,570
- *Gain (Sss,,,, Wind)* = 970 - (215) 1.0 - *(315)* ,918 = ,019

The partially learned decision tree resulting from the first step of ID3. The training examples are sorted to the corresponding descendant nodes. The *Overcast* descendant has only positive examples and therefore becomes a leaf node with classification *Yes.* The other two nodes will be further expanded, by selecting the attribute with highest information gain relative to the new subsets of examples that guides this hill-climbing search is the information gain measure. By viewing ID^ in terms of its search space and search strategy, we can get some insight into its capabilities and limitations.

1 ~ 3 ' shypothesis space of all decision trees is a *complete* space of finite discrete-valued functions, relative to the available attributes. Because every finite discrete-valued function can be represented by some decision tree, ID3

avoids one of the major risks of methods that search incomplete hypothesis spaces (such as methods that consider only conjunctive hypotheses): that the hypothesis space might not contain the target function.

ID3 maintains only a single current hypothesis as it searches through the space of decision trees. This contrasts, for example, with the earlier ver-sion space c a n d i d a t e - ~ l i r n i n a t - o d , which maintains the set of *all* hypotheses consistent with the available training examples. By determining only a single hypothesis, ID^ loses the capabilities that follow from explicitly representing all consistent hypotheses. For example, it does not have the ability to determine how many alternative decision trees are consistent with the available training data, or to pose new instance queries that optimally resolve among these competing hypotheses.

ID3 in its pure form performs no backtracking in its search. Once it, selects an attribute to test at a particular level in the tree, it never backtracks to reconsider this choice. Therefore, it is susceptible to the usual risks of hill-climbing search without backtracking: converging to locally optimal solutions that are not globally optimal. In the case of ID3, a locally optimal solution corresponds to the decision tree it selects along the single search path it explores. However, this locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search. Below we discuss an extension that adds a form of backtracking (post-pruning the decision tree).

ID3 uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis. This contrasts with methods that make decisions incrementally, based on individual training examples (e.g., FIND-Sor CANDIDATE-ELIMINATION) One advantage. of using statistical properties of all the examples (e.g., information gain) is that

the resulting search is much less sensitive to errors in individual training examples. ID3 can be easily extended to handle noisy training data by

modifying its termination criterion to accept hypotheses that imperfectly fit the training data.

INDUCTIVE BIAS IN DECISION TREE LEARNING

What is the policy by which ID3 generalizes from observed training examples to classify unseen instances? In other words, what is its inductive bias? Recall from Chapter 2 that inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances.

Given a collection of training examples, there are typically many decision trees consistent with these examples. Describing the inductive bias of ID3 there-fore consists of describing the basis by which it chooses one of these consis-tent hypotheses over the others. Which of these decision trees does ID3 choose? It chooses the first acceptable tree it encounters in its simple-to-complex, hill-climbing search through the space of possible trees. Roughly speaking, then, the ID3 search strategy (a) selects in favor of shorter trees over longer ones, and selects trees that place the attributes with highest information gain closest to the root. Because of the subtle interaction between the attribute selection heuristic used by ID3 and the particular training examples it encounters, it is difficult to characterize precisely the inductive bias exhibited by ID3. However, we can approximately characterize its bias as a preference for short decision trees over complex trees.

Approximate inductive bias of ID3: Shorter trees are preferred over larger trees.

In fact, one could imagine an algorithm similar to ID3 that exhibits precisely this inductive bias. Consider an algorithm that begins with the empty tree and

searches *breadth Jirst* through progressively more complex trees, first considering all trees of depth 1, then all trees of depth 2, etc. Once it finds a decision tree consistent with the training data, it returns the smallest consistent tree at that search depth (e.g., the tree with the fewest nodes). Let us call this breadth-first search algorithm BFS-ID3. BFS-ID3 finds a shortest decision tree and thus exhibits precisely the bias "shorter trees are preferred over longer trees." ID3 can be viewed as an efficient approximation to BFS-ID3, using a greedy heuristic search to attempt to find the shortest tree without conducting the entire breadth-first search through the hypothesis space.

Because ID3 uses the information gain heuristic and a hill climbing strategy, it exhibits a more complex bias than BFS-ID3. In particular, it does not always find the shortest consistent tree, and it is biased to favor trees that place attributes with high information gain closest to the root. A closer approximation to the inductive bias of ID3: Shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

Restriction Biases and Preference Biases

There is an interesting difference between the types of inductive bias exhibited by ID3 and by the CANDIDATE-ELIMINATION algorithm discussed in Chapter 2.

Consider the difference between the hypothesis space searches in these two approaches:

- ID3 searches a complete hypothesis space (i.e., one capable of expressing any finite discrete-valued function). It searches incompletely through this space, from simple to complex hypotheses, until its termination condition is met (e.g., until it finds a hypothesis consistent with the data). Its inductive bias

is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias.

- The version space CANDIDATE-ELIMINATION algorithm searches an incomplete hypothesis space (i.e., one that can express only a subset of the potentially teachable concepts). It searches this space completely, finding every hypothesis consistent with the training data. Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. Its search strategy introduces no additional bias.

In brief, the inductive bias of ID3 follows from its search strategy, whereas the inductive bias of the CANDIDATE-ELIMINATION algorithm follows from the definition of its search space.

The inductive bias of ID3 is thus a preference for certain hypotheses over others (e.g., for shorter hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is typically called a preference bias (or, alternatively, a search bias). In contrast, the bias of the CANDIDATE-ELIMINATION algorithm is in the form of a categorical restriction on the set of hypotheses considered. This form of bias is typically called a restriction bias (or, alternatively, a language bias).

Given that some form of inductive bias is required in order to generalize beyond the training data (see Chapter 2), which type of inductive bias shall we prefer; a preference bias or restriction bias?

Typically, a preference bias is more desirable than a restriction bias, be-cause it allows the learner to work within a complete hypothesis space that is assured to contain the unknown target function. In contrast, a restriction bias that strictly limits the set of potential hypotheses is generally less desirable, because it introduces the possibility of excluding the unknown target function

altogether.

Whereas ID3 exhibits a purely preference bias and CANDIDATE-ELIMINATION a purely restriction bias, some learning systems combine both. Consider, for example, the program described ,for learning a numerical evaluation function for game playing. In this case, the learned evaluation function is represented by a linear combination of a fixed set of board features, and the learning algorithm adjusts the parameters of this linear combination to best fit the available training data. In this case, the decision to use a linear function to represent the evaluation function introduces a restriction bias (nonlinear evaluation functions cannot be represented in this form). At the same time, the choice of a particular parameter tuning method (the LMS algorithm in this case) introduces a preference bias stem-ming from the ordered search through the space of all possible parameter values.

Why Prefer Short Hypotheses?

Is ID3's inductive bias favoring shorter decision trees a sound basis for generalizing beyond the training data? Philosophers and others have debated this question for centuries, and the debate remains unresolved to this day. William of Occam was one of the first to discuss the question, around the year 1320, so this bias often goes by the name of Occam's razor.

Occam's razor: Prefer the simplest hypothesis that fits the data.

Of course giving an inductive bias a name does not justify it. Why should one prefer simpler hypotheses? Notice that scientists sometimes appear to follow this inductive bias. Physicists, for example, prefer simple explanations for the motions of the planets, over more complex explanations. Why? One argument is that because there are fewer short hypotheses than long ones (based on straightforward combinatorial arguments), it is less likely that one will find a short hypothesis that coincidentally fits the training data. In contrast there are

often many very complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data. Consider decision tree hypotheses, for example. There are many more 500-node decision trees than 5-node decision trees. Given a small set of 20 training examples, we might expect to be able to find many 500-node decision trees consistent with these, whereas we would be more surprised if a 5-node decision tree could perfectly fit this data. We might therefore believe the 5-node tree is less likely to be a statistical coincidence and prefer this hypothesis over the 500-node hypothesis.

Upon closer examination, it turns out there is a major difficulty with the above argument. By the same reasoning we could have argued that one should prefer decision trees containing exactly 17 leaf nodes with 11 nonleaf nodes, that use the decision attribute A1 at the root, and test attributes A2 through A l l , in numerical order. There are relatively few such trees, and we might argue (by the same reasoning as above) that our a priori chance of finding one consistent with an arbitrary set of data is therefore small. The difficulty here is that there are very many small sets of hypotheses that one can define-most of them rather arcane. Why should we believe that the small set of hypotheses consisting of decision trees with *short descriptions* should be any more relevant than the multitude of other small sets of hypotheses that we might define?

A second problem with the above argument for Occam's razor is that the size of a hypothesis is determined by the particular representation used internally by the learner. Two learners using different internal representations could therefore naive at different hypotheses, both justifying their contradictory conclusions by Occam's razor! For example, the function represented by the learned decision tree in Figure 3.1 could be represented as a tree with just one decision node, by a learner that uses the Boolean attribute XYZ, where we

define the attribute XYZ to be true for instances that are classified positive by the decision tree and false otherwise. Thus, two learners, both applying Occam's razor, would generalize in different ways if one used the XYZ attribute to describe its examples and the other used only the attributes Outlook, Temperature, Humidity, and Wind.

This last argument shows that Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners that perceive these examples in terms of different internal representations. On this basis we might be tempted to reject Occam's razor altogether. However, consider the following scenario that examines the question of which internal representations might arise from a process of evolution and natural selection. Imagine a population of artificial learning agents created by a simulated evolutionary process involving reproduction, mutation, and natural selection of these agents. Let us assume that this evolutionary process can alter the perceptual systems of these agents from generation to generation, thereby changing the internal attributes by which they perceive their world. For the sake of argument, let us also assume that the learning agents employ a fixed learning algorithm (say ID3) that cannot be altered by evolution. It is reasonable to assume that over time evolutions will produce internal representation that make these agents increasingly successful within their environment. Assuming that the success of an agent depends highly on its ability to generalize accurately, we would therefore expect evolution to develop internal representations that work well with whatever learning algorithm and inductive bias is present. If the species of agents employs a learning algorithm whose inductive bias is Occam's razor, then we expect evolution to produce internal representations for which Occam's razor is a successful strategy. The essence of the argument here is that evolution

will create internal representations that make the learning algorithm's inductive bias a self-fulfilling prophecy; simply because it can alter the representation easier than it can alter the learning algorithm.

For now, we leave the debate regarding Occam's razor. We will revisit it in Chapter 6, where we discuss the Minimum Description Length principle, a version of Occam's razor that can be interpreted within a Bayesian framework.

ISSUES IN DECISION TREE LEARNING

Practical issues in learning decision trees include determining how deeply to grow the decision tree, handling continuous attributes, choosing an appropriate attribute selection measure, and training data with missing attribute values, handling attributes with differing costs, and improving computational efficiency. Below we discuss each of these issues and extensions to the basic ID3 algorithm that address them. ID3 has itself been extended to address most of these issues, with the resulting system renamed C4.5 (Quinlan 1993).

UNIT-II

LINEAR DISCRIMINANTS

Perceptron (MLP): Going forwards, backwards, MLP in practices, deriving back; Propagation support vector Machines: Optimal separation, kernels.

Perceptron (MLP): Going forwards, backwards, MLP in practices, deriving back; Propagation support vector Machines: Optimal separation, kernels.

Multilayer networks of such units and consider several general issues such as the representational capabilities of ANNs, nature of the hypothesis space search, over-fitting problems, and alternatives to the BACKPROPAGATION algorithm. A detailed example is also presented applying BACKPROPAGATION to face recognition, and directions are provided for the reader to obtain the data and code to experiment further with this application.

PERCEPTRONS

One type of ANN system is based on a unit called a Perceptron, A Perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs *xl* through *x,,* the output *o(x1, . . . ,x,)* computed by the Perceptron is

$$o(x1,\ldots,x,) = \begin{cases} 1 \text{ if } wo + w\,l\,x\,l + \sim 2\,x + 2 - . + W, X, > 0 \\ -1 \text{ otherwise} \end{cases}$$

Where each *wi* is a real-valued constant, or weight, that determines the contribution of input *xi* to the Perceptron output. Notice the quantity *(- wO)* is a threshold that the weighted combination of inputs *wlxl* + . . . + *wnxn* must surpass in order for the Perceptron to output a 1.

To simplify notation, we imagine an additional constant input *xo* = 1, allowing us to write the above inequality as *C: =o wixi* > 0, or in vector form as *iir* ..i! > 0. For brevity, we will sometimes write the Perceptron function as Learning a Perceptron involves choosing values for the weights *wo* . . . *w,*. Therefore, the space H of candidate hypotheses considered in Perceptron learning is the set of all possible real-valued weight vectors.

Representational Power of Perceptron:

We can view the Perceptron as representing a hyper plane decision surface in the n-dimensional space of instances (i.e., points). The Perceptron outputs a 1 for instances lying on one side of the hyper plane and outputs a -1 for instances lying on the other side, as illustrated in Figure 4.3. The equation for this decision hyper plane is *iir* . .i! = 0. Of course, some sets of positive and negative examples cannot be separated by any hyper plane. Those that can be separated are called linearly separable sets of examples.

A single Perceptron can be used to represent many Boolean functions. For example, if we assume Boolean values of 1 (true) and -1 (false), then one way to use a two-input Perceptron to implement the AND function is to set the weights *wo* = *-3,* and *wl* = *wz* = *.5.* This Perceptron can be made to represent the OR function instead by altering the threshold to *wo* = -.3. In fact, AND and OR can be viewed as special cases of m-of-n functions: that is, functions where at least

of the n inputs to the Perceptron must be true. The OR function corresponds to *rn* = 1 and the AND function to m = n. Any m-of-n function is easily represented using a Perceptron by setting all input weights to the same value (e.g., 0.5) and then setting the threshold *wo* accordingly.

Perceptions can represent all of the primitive Boolean functions AND, OR, NAND (1 AND), and NOR (1 OR). Unfortunately, however, some Boolean functions cannot be represented by a single Perceptron, such as the XOR function whose value is 1 if and only if xl # xz. Note the set of linearly no separable training examples shown corresponds to this XOR function.

The ability of perceptions to represent AND, OR, NAND, and NOR is important because *every* Boolean function can be represented by some network of interconnected units based on these primitives. In fact, every Boolean function can be represented by some network of perceptions only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage. One way is to represent the Boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations). Note that the input to an AND Perceptron can be negated simply by changing the sign of the corresponding input weight.

Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single Perceptron. Here the precise learning problem is to determine a weight vector that causes the Perceptron to produce the correct f 1 output for each of the given training examples.

Several algorithms are known to solve this learning problem. Here we consider two: the Perceptron rule and the delta rule (a variant of the LMS rule used in Chapter 1 for learning evaluation functions). These two algorithms are guaran-teed to converge to somewhat different acceptable hypotheses, under somewhat different conditions. They are important to ANNs because they provide the basis for learning networks of many units.

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the Perceptron to each training example, modify-ing the Perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the Perceptron classifies all training examples correctly. Weights are modified at each step according to the Perceptron training rule, which revises the weight *wi* associated with input *xi* according to the rule Here *t* is the target output for the current training example, *o* is the output generated by the Perceptron, and *q* is a positive constant called the *learning rate.* The role of the learning rate is to moderate the degrees to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

Why should this update rule converge toward successful weight values? To get an intuitive feel, consider some specific cases. Suppose the training example is correctly classified already by the Perceptron. In this case, *(t - o)*

is zero, making *Awi* zero, so that no weights are updated. Suppose the Perceptron outputs a -1, when the target output is +1. To make the Perceptron output a + 1 instead of - 1 in this case, the weights must be altered to increase the value of *G . 2* . For example, if *xi* r *0,*then increasing *wi* will bring the Perceptron closer to correctly classifying this example. Notice the training rule will increase *w,* in this case, because *(t - o), 7 ,* and *Xi* are all positive. For example, if *xi = .8, q = 0.1, t = 1* , and *o = - 1* , then the weight update will be *Awi = q(t - o)xi = O . 1 (1 - (-1))0.8 = 0.16.* On the other hand, if *t = - 1* and *o* = 1, then weights associated with positive *xi* will be decreased rather than increased.

In fact, the above learning procedure can be proven to converge within a finite number of applications of the Perceptron training rule to a weight vector that correctly classifies all training examples, *provided the training examples are linearly separable* and provided a sufficiently small 7 is used. If the data are not linearly separable, convergence is not assured.

Gradient Descent and the Delta Rule

Although the Perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable. A second training rule, called the *delta rule,* is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The key idea behind the delta rule is to use *gradient descent* to search the hypothesis space of possible weight vectors to find the weights that best fit the train-ing examples. This rule is important because gradient descent provides

the basis for the BACKPROPAGATION algorithm, which can learn networks with many inter-connected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces contain-ing many different types of continuously parameterized hypotheses. Thus, a linear unit corresponds to the first stage of a Perceptron, without the threshold.

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the *training error* of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is we assume these are fixed during training, so we do not bother to write E as an explicit function of these. Bayesian justification for choosing this particular definition of E. In particular, there we show that under certain conditions the hypothesis that minimizes E is also the most probable hypothesis in H given the training data.

VISUALIZING THE HYPOTHESIS SPACE

To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values, Here the axes *wo* and *w l* represent possible values for the two weights of a simple linear unit. The *wo, w l* plane therefore represents the entire hypothesis space. The vertical axis indicates the error E relative to some fixed set of training examples. The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space (we desire a hypothesis with minimum error). Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a

single global minimum. The specific parabola will depend, of course, on the particular set of training examples.

Gradient descent search determines a weight vector that minimizes *E* by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface. This process continues until the global minimum error is reached.

DERIVATION OF THE GRADIENT DESCENT RULE

How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of *E* with respect to each component of the vector *2* . This vector derivative is called the *gradient* of *E* with respect to *221,* written ~ ~ *(i i r)* .

Notice *VE(221)*is itself a vector, whose components are the partial derivatives of *E* with respect to each of the *wi.* When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E . The negative of this vector therefore gives the direction of steepest decrease. For example, the arrow in shows the negated gradient - *VE(G)* for a particular point in the *wo,wl* plane. Since the gradient specifies the direction of steepest increase of *E,* the training rules for gradient descent.

The negative sign is present because we want to move the weight vector in the direction that *decreases* E. This training rule can also be written in its component form which makes it clear that steepest descent is achieved by altering each component *w,*of ii in proportion to E.

To summarize, the gradient descent algorithm for training linear units is as follows: Pick an initial random weight vector. Apply the linear unit to all training examples, and then compute Awi for each weight according to Update each weight *wi* by adding Awi, then repeat this process. Because the error

surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate *q* is used. If *r)* is too large; the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to gradually reduce the value of *r)* as the number of gradient descent steps grows.

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

One common variation on gradient descent intended to alleviate these difficulties is called incremental gradient descent, or alternatively stochastic gradient descent. Whereas the gradient descent training rule presented and computes weight updates after summing over *a22* the training examples in D, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example. In standard gradient descent, the error is

summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example. Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent *r,* In cases where there are multiple local minima with respect to *E ($* , stochas-tic gradient descent can sometimes avoid falling into these local minima because it uses the various *V E d (G)*rather than *V E (6)* to guide its search. Both stochastic and standard gradient descent methods are commonly used in practice.

Remarks

We have considered two similar algorithms for iteratively learning Perceptron weights. The key difference between these algorithms is that the Perceptron training rule updates weights based on the error in the threshold Perceptron output, whereas the delta rule updates weights based on the error in the unthresholded linear combination of inputs.

The difference between these two training rules is reflected in different convergence properties. The Perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable. The delta rule converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable. A detailed presentation of the convergence proofs can be found in Hertz et al. (1991).

Another possible algorithm for learning the weight vector is linear programming. Linear programming is a general, efficient method for solving sets of linear inequalities. Notice each training example corresponds to an

inequality of the form zZI - x'> 0 or *G* . x'*5* 0, and their solution is the desired weight vector. Un-fortunately, this approach yields a solution only when the training examples are linearly separable; however, Duda and Hart (1973, p. 168) suggest a more subtle formulation that accommodates the no separable case. In any case, the approach of linear programming does not scale to training multilayer networks, which is our primary concern. In contrast, the gradient descent approach, on which the delta rule is based, can be easily extended to multilayer networks, as shown in the following section.

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

As noted single perceptions can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the BACKPROPA-CATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces. For example, a typical multilayer network and decision surface is depicted. Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., "hid," "had," "head," "hood," etc.). The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space. As shown in the figure, it is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces of single units shown earlier. This section discusses how to learn such multilayer networks using a gradient descent algorithm similar to that discussed in the previous section.

A Differentiable Threshold Unit

What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the previous Section, for which we have already derived a gradient descent learning rule. How-ever, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The Perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the sigmoid unit-a unit very much like a Perceptron, but based on a smoothed, differentiable threshold function.

The sigmoid unit first computes a linear combination of its inputs, and then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a Continuous function of its input. More precisely, the sigmoid unit computes its output *o* as a is often called the sigmoid function or, alternatively, the logistic function. Note its output ranges between 0 and 1, increasing monotonically with its input because it maps a very large input domain to a small range of outputs; it is often referred to as the squashing function of the unit. The sigmoid function has the useful property that its derivative is easily we shall see, the gradient descent learning rule makes use of this derivative. Other differentiable functions with easily calculated derivatives are sometimes used in place of a. For example, the term *e-y* in the sigmoid function definition is sometimes replaced by *e-k'y* where k is some positive constant that determines the steepness of the threshold. The function is also sometimes used in place of the sigmoid function.

THE BACKPROPAGATION ALGORITHM

The BACKPROPAGATION algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units where outputs is the set of output units in the network. The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by all possible weight values for all the units in the network. The situation can be visualized in terms of an error surface similar to that shown for linear units in. The error in that diagram is replaced by our new definition of E, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E.The stochastic gradient descent version of the BACKPROPAGATION algorithm for feed forward networks containing two layers of sigmoid units.

One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface. Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error. Despite this obstacle, in practice BACKPROPAGATION has been found to produce excellent results in many real-world applications.

The algorithm as described here applies to layer feed forward networks containing two layers of sigmoid units, with units at each layer connected to

all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of BACK-PROPAGATION. An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network *xji* denotes the input from node i to unit *j* , and *wji* denotes the corresponding weight.

Notice the algorithm in begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, and then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

Presentation of each training example. This corresponds to a stochastic approximation to gradient descent. To obtain the true gradient of E one would sum the 6, *x,* values over all training examples before altering weight values. The weight-update loop in BACKPROPAGATION may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure. One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold, or once the error on a separate validation set of examples meets some criterion. The choice of termination criterion is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to over fitting the training data.

ADDING MOMENTUM

Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. Perhaps the most common is to alter the weight-update rule in the algorithm by making the weight update on the nth iteration depend partially on the update that occurred during the (n - 1) th iteration, as follows: Here Awji(n) is the weight update performed during the nth iteration through the main loop of the algorithm, and 0 *5* a < 1 is a constant called the momentum. To see the effect of this momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentum less) ball rolling down the error surface. The effect of *a!* is to add momentum that tends to keep the ball rolling in the same direction from one iteration to the next. This can sometimes have the effect of keeping the ball rolling through small local minima in the error surface, or along flat regions in the surface where the ball would stop if there were no momentum. It also has the effect of gradually increasing the step size of the search in regions where the gradient is unchanging, thereby speeding convergence.

LEARNING IN ARBITRARY ACYCLIC NETWORKS

The definition of BACKPROPAGATION presented applies to two-layer networks. However, the algorithm given their easily generalizes to feed forward networks of arbitrary depth. The weight update rule seen in is retained, and the only change is to the procedure for computing 6 values. In general, the *6,* value for a unit *r* in layer *rn* is computed from the *6* values at the next deeper layer *rn* + 1 according. so all we are really saying here is that this step may be repeated for any number of hidden layers in the network.

It is equally straightforward to generalize the algorithm to any directed acyclic graph, regardless of whether the network units are arranged in uniform layers

as we have assumed up to now. In the case that they are not, the rule for calculating *6* for any internal unit (i.e., any unit that is not an output) is where Downstream(r) is the set of units immediately downstream from unit *r* in the network: that is, all units whose inputs include the output of unit r.

Derivation of the BACKPROPAGATION Rule

This section presents the derivation of the BACKPROPAGATION weight-tuning rule. It may be skipped on a first reading, without loss of continuity. The specific problem we address here is deriving the stochastic gradient descent rule implemented by the algorithm. Recall from that stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error *Ed* with respect to this single example. In other words, for each training example d every weight *wji* is updated by adding to its *Awji* where *Ed* is the error on training example d, summed overall output units in the network, Here outputs is the set of output units in the network, *tk* is the target value of unit k for training example d, and *ok* is the output of unit k given training example d.

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables. We will follow the notation a subscript *j* to denote to the jth unit of the network as follows:

xji = the ith input to unit *j*

wji = the weight associated with the ith input to unit *j netj* = *xi wjixji* (the weighted sum of inputs for unit *j) oj* = the output computed by unit *j*

t, = the target output for unit *j*= the sigmoid function *outputs* = the set of units in the final layer of the network *Downstream(j)* = the set of units whose immediate inputs include the output of unit *j*

We consider two cases in turn: the case where unit j is an output unit for the network, and the case where j is an internal unit.

Case 1: raini in^ Rule for Output Unit Weights. Just as *wji* can influence the rest of the network only through *net,, net,* can influence the network only through *o j* . Therefore, we can invoke the chain rule again to write

To begin, consider just the first term in Equation

The derivatives *& (tk - ok12* will be zero for all output units k except when k = j.

We therefore drop the summation over output units and simply set k = j.

Next consider the second term in Equation. Since *oj = a(netj)* ,the derivative $ is just the derivative of the sigmoid function, which we have already noted is equal to *a(netj)(l - a(netj))* .

Case 2: Training Rule for Hidden Unit Weights. In the case where *j* is an internal, or hidden unit in the network, the derivation of the training rule for *wji* must take into account the indirect ways in which *wji* can influence the network outputs and hence *Ed.* For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network (i.e., all units whose direct inputs include the output of unit j). We denote this set of units by *Downstream (j*). Notice that *netj* can influence the network outputs (and therefore *E d)* only through the units in *Downstream (j).* Therefore, we can write Rearranging terms and using *S j* to denote.

REMARKS ON THE BACKPROPAGATIONALGORITHM

Convergence and Local Minima

As shown above, the BACKPROPAGATION algorithm implements a gradient descent search through the space of possible network weights, iteratively reducing the error E between the training example target values and

the network outputs. Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these. As a result, BACKPROPAGATION over multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.

Despite the lack of assured convergence to the global minimum error, BACK-PROPAGATION is a highly effective function approximation method in practice. In many practical applications the problem of local minima has not been found to be as severe as one might fear. To develop some intuition here, consider that networks with large numbers of weights correspond to error surfaces in very high dimensional spaces (one dimension per weight). When gradient descent falls into a local minimum with respect to one of these weights, it will not necessarily be in a local minimum with respect to the other weights. In fact, the more weights in the network, the more dimensions that might provide "escape routes" for gradient descent to fall away from the local minimum with respect to this single weight.

A second perspective on local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases. Notice that if network weights are initialized to values near zero, then during early gradient descent steps the network will represent a very smooth function that is approximately linear in its inputs. This is because the sigmoid threshold function itself is approximately linear when the weights are close to zero .Only after the weights have had time to grow will they reach a point where they can represent highly nonlinear network functions. One might expect more local minima to exist in the region of the weight space that represents these more complex functions.

One hopes that by the time the weights reach this point they have already moved close enough to the global minimum that even local minima in this region are acceptable.

Despite the above comments, gradient descent over the complex error surfaces represented by ANNs is still poorly understood, and no methods are known to predict with certainty when local minima will cause difficulties. Common heuristics to attempt to alleviate the problem of local minima include: Momentum can sometimes carry the gradient descent procedure through narrow local minima (though in principle it can also carry it through narrow global minima into other local minima!). Use stochastic gradient descent rather than true gradient descent. The stochastic approximation to gradient descent effectively descends a different error surface for each training example.

UNIT-III

BASIC STATISTICS

Bayesian learning: Introduction, Bayes theorem, Bayes optimal classifier, naïve Bayes classifier. Graphical models: Bayesian networks, approximate inference, making Bayesian networks, hidden Markov models, the forward algorithm.

Mean and Variance

Two properties of a random variable that are often of interest are its expected value (also called its mean value) and its variance. The expected value is the average of the values taken on by repeatedly sampling the random variable.

Definition: Consider a random variable *Y* that takes on the possible values yl, . *yn.* The expected value of *Y , E [Y] ,* For example, if *Y* takes on the value 1 with probability .7 and the value 2 with probability .3, then its expected value is (1 .0.7 + 2.0.3 = 1.3). In case the random variable *Y* is governed by a Binomial distribution, then it can be shown that In case the random variable Y is governed by a Binomial distribution, then the variance and standard deviation.

Estimators, Bias, and Variance

Now that we have shown that the random variable error (h) obeys a Binomial distribution, we return to our primary question: What is the likely difference between *errors (h)* and the true error *(h)?,* Statisticians call *errors (h)* an *estimator* for the true error *errorv (h).* In general, an estimator is any random variable used to estimate some parameter of the underlying population from which the sample is drawn. An obvious question to ask about any estimator is

whether on average it gives the right estimate. We define the estimation bias to be the difference between the expected value of the estimator and the true value of the parameter.

If the estimation bias is zero, we say that Y is an *unbiased estimator* for *p.* Notice this will be the case if the average of many random values of Y generated by repeated random experiments (i.e., E[Y]) converges toward p.

Is *errors (h)* an unbiased estimator for *errorv(h)?* Yes, because for a Binomial distribution the expected value of *r* is equal to *np* (Equation r5.41). It follows; given that *n* is a constant, that the expected value of *rln* is p.

Two quick remarks are in order regarding the estimation bias. First, when we mentioned at the beginning of this chapter that testing the hypothesis on the training examples provides an optimistically biased estimate of hypothesis error, it is exactly this notion of estimation bias to which we were referring. In order for *errors(h)* to give an unbiased estimate of *errorv(h),*the hypothesis *h* and sample must be chosen independently. Second, this notion of estimation bias should not be confused with the *inductive bias* of a learner introduced in Bayesian reasoning provides a probabilistic approach to inference. It is based on the assumption that the quantities of interest are governed by probability distributions and that optimal decisions can be made by reasoning about these probabilities together with observed data. It is important to machine learning because it provides a quantitative approach to weighing the evidence supporting alternative hypotheses. Bayesian reasoning provides the basis for learning algorithms that directly manipulate probabilities, as well as a framework for analyzing the operation of other algorithms that do not explicitly manipulate probabilities.

INTRODUCTION

Bayesian learning methods are relevant to our study of machine learning for two different reasons. First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems. For example, Michie et al. (1994) provide a detailed study comparing the naive Bayes classifier to other learning algorithms, including decision tree and neural network algorithms. These researchers show that the naive Bayes classifier is competitive with these other learning algorithms in many cases and that in some cases it outperforms these other methods. In this chapter we describe the naive Bayes classifier and provide a detailed example of its use. In particular, we discuss its application to the problem of learning to classify text documents such as electronic news articles.For such learning tasks, the naive Bayes classifier is among the most effective algorithms known.

The another reason that Bayesian methods are important to our study of machine learning is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities. For example, in this chapter we analyze algorithms such as the FIND-S and CANDIDATE-ELIMINATION algorithms of Chapter 2 to determine conditions under which they output the most probable hypothesis given the training data. We also use a Bayesian analysis to justify a key design choice in neural network learning algorithms: choosing to minimize the sum of squared errors when searching the space of possible neural networks. We also derive an alternative error function, cross entropy, that is more appropriate than sum of squared errors when learn-ing target functions that predict probabilities. We use a Bayesian perspective to analyze the inductive bias of decision tree learning algorithms that favor short decision trees and examine the closely

related Minimum Description Length principle. A basic familiarity with Bayesian methods is important to understanding and characterizing the operation of many algorithms in machine learning. Features of Bayesian learning methods include:

Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct. This provides a more flexible approach to learning than algorithms that completely eliminate a hypothesis if it is found to be inconsistent with any single example.

Prior knowledge can be combined with observed data to determine the final probability ~f a hypothesis. In Bayesian learning, prior knowledge is provided by asserting (1) a prior probability for each candidate hypothesis, and (2) A probability distribution over observed data for each possible hypothesis.

Bayesian methods can accommodate hypotheses that make probabilistic predictions (e.g., hypotheses such as "this pneumonia patient has a 93% chance of complete recovery"). New instances can be classified by combining the predictions of multiple hypotheses, weighted by their probabilities.

Even in cases where Bayesian methods prove computationally intractable, they can provide a standard of optimal decision making against which other practical methods can be measured.

One practical difficulty in applying Bayesian methods is that they typically require initial knowledge of many probabilities. When these probabilities are not known in advance they are often estimated based on background knowledge, previously available data, and assumptions about the form of the underlying distributions. A second practical difficulty is the significant computational cost required to determine the Bayes optimal hypothesis in the general case (linear in the number of candidate hypotheses). In certain specialized situations, this computational cost can be significantly reduced.

Bayes theorem and defines maximum likelihood and maximum a posteriori probability hypotheses. The four subsequent sections then apply this probabilistic framework to analyze several issues and learning algorithms discussed in earlier chapters. For example, we show that several previously described algorithms out-put maximum likelihood hypotheses, under certain assumptions. The remaining sections then introduce a number of learning algorithms that explicitly manipulate probabilities. These include the Bayes optimal classifier, Gibbs algorithm, and naive Bayes classifier. Finally, we discuss Bayesian belief networks, a relatively recent approach to learning based on probabilistic reasoning, and the EM algorithm, a widely used algorithm for learning in the presence of unobserved variables.

BAYES THEOREM

In machine learning we are often interested in determining the best hypothesis from some space H, given the observed training data D. One way to specify what we mean by the *best* hypothesis is to say that we demand the *most probable* hypothesis, given the data *D* plus any initial knowledge about the prior probabilities of the various hypotheses in H. Bayes theorem provides a direct method for calculating such probabilities. More precisely, Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

To define Bayes theorem precisely, let us first introduce a little notation. We shall write $P(h)$ to denote the initial probability that hypothesis h holds, before we have observed the training data. $P(h)$ is often called the *prior probability* of h and may reflect any background knowledge we have about the chance that h is a correct hypothesis. If we have no such prior knowledge, then

we might simply assign the same prior probability to each candidate hypothesis. Similarly, we will write *P (D)* to denote the prior probability that training data D will be observed (i.e., the probability of D given no knowledge about which hypothesis holds). Next, we will write *P(D1h)* to denote the probability of observing data *D* given some world in which hypothesis *h* holds. More generally, we write *P(xly)* to denote the probability of x given *y*. In machine learning problems we are interested in the probability *P (h1 D)* that *h* holds given the observed training data *D* . *P (h1 D)* is called the *posterior probability* of *h* , because it reflects our confidence that *h* holds after we have seen the training data *D* . Notice the posterior probability *P(h1D)* reflects the influence of the training data *D* , in contrast to the prior probability *P(h),* which is independent of D.

Bayes theorem is the cornerstone of Bayesian learning methods because it provides a way to calculate the posterior probability *P(hlD),* from the prior probability *P(h),* together with *P (D)* and *P (D (h)* .

Bayes theorem:

As one might intuitively expect, *P(h ID)* increases with *P(h)* and with *P(D1h)* according to Bayes theorem. It is also reasonable to see that *P(hl D)* decreases as *P (D)* increases, because the more probable it is that *D* will be observed independent of *h* , the less evidence *D* provides in support of *h.*

In many learning scenarios, the learner considers some set of candidate hypotheses H and is interested in finding the most probable hypothesis *h* E H given the observed data *D* (or at least one of the maximally probable if there are several). Any such maximally probable hypothesis is called a maximum a posteriori (MAP) hypothesis. We can determine the MAP hypotheses by using Bayes theorem to calculate the posterior probability of each candidate

hypothesis. More precisely, we will say that *MAP* is a MAP hypothesis provided

Notice in the final step above we dropped the term *P (D)* because it is a constant independent of *h* .

In some cases, we will assume that every hypothesis in H is equally probable a priori *(P (h i) = P(h;)* for all *hi* and *h;* in H). In this case we can further simplify Equation (6.2) and need only consider the term *P(D1h)* to find the most probable hypothesis. *P(Dlh)* is often called the *likelihood* of the data *D* given *h,* and any hypothesis that maximizes *P(Dlh)* is called a *maximum likelihood* (ML) hypothesis, *hML* .

hML= argmax *P(Dlh)*

h € H

In order to make clear the connection to machine learning problems, we introduced Bayes theorem above by referring to the data *D* as training examples of some target function and referring to H as the space of candidate target functions. In fact, Bayes theorem is much more general than suggested by this discussion. It can be applied equally well to any set H of mutually exclusive propositions whose probabilities sum to one (e.g., "the sky is blue," and "the sky is not blue"). In this chapter, we will at times consider cases where H is a hypothesis space containing possible target functions and the data *D* are training examples. At other times we will consider cases where H is some other set of mutually exclusive propositions, and *D* is some other kind of data.

To illustrate Bayes rule, consider a medical diagnosis problem in which there are

Two alternative hypotheses: (1) that the patient; (2) that the patient does not. The available data is from a particular laboratory test with two possible outcomes: $ (positive) and *8* (negative). We have prior knowledge that over the entire population of people only . Furthermore, the lab test is only an imperfect indicator of the disease. The test returns a correct positive result in only 98% of the cases in which the disease is actually present and a correct negative result in only 97% of the cases in which the disease is not present. In other cases, the test returns the opposite result. The above situation can be summarized by the following probabilities:

Suppose we now observe a new patient for whom the lab test returns a positive result. Should we diagnose the patient as having cancer or not? Thus, *h* ~ = -~cancer*p*. The exact posterior hob abilities can also be determined by normalizing the above quantities so that they sum to 1 (e.g., P(cancer($) = .00;~~298= .21). This step is warranted because Bayes theorem states that the posterior probabilities are just the above quantities divided by the probability of the data, P(@). Although P($) was not provided directly as part of the problem statement, we can calculate it in this fashion because we know that P(cancerl$) and P(-cancerl$) must sum to 1 (i.e., either the patient has cancer or they do not). Notice that while the posterior probability of cancer is significantly higher than its prior probability, the most probable hypothesis is still that the patient does not have cancer.

As this example illustrates, the result of Bayesian inference depends strongly on the prior probabilities, which must be available in order to apply the method directly. Note also that in this example the hypotheses are not completely accepted or rejected, but rather become more or less probable as more data is observed.

BAYES THEOREM AND CONCEPT LEARNING

What is the relationship between Bayes theorem and the problem of concept learning? Since Bayes theorem provides a principled way to calculate the posterior probability of each hypothesis given the training data, we can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, and then outputs the most probable. As we shall see, one interesting result of this comparison is that under certain conditions several algorithms discussed in earlier chapters output the same hypotheses as this brute-force Bayesian algorithm, despite the fact that they do not explicitly manipulate probabilities and are considerably more efficient.

Brute-Force Bayes Concept Learning

Consider the concept learning problem first introduced. In particular, assume the learner considers some finite hypothesis space H defined over the instance space X, in which the task is to learn some target concept c : X + {0,1}. As usual, we assume that the learner is given some sequence of training examples *((x ~d,l). . . (xm,d m))*where *xi* is some instance from X and where *di* is the target value of *xi* (i.e., *di = c(xi))* .To simplify the discussion in this section, we assume the sequence of instances *(xl . . .xm)*is held fixed, so that the training data D can be written simply as the sequence of target values D = *(dl . . .d m)* .

BRUTE-FORCEMAP LEARNING algorithm

For each hypothesis *h* in H, calculate the posterior probability Output of the hypothesis *hMAP*with the highest posterior probability

This algorithm may require significant computation, because it applies Bayes theorem to each hypothesis in H to calculate *P(hJD)* . While this may prove

impractical for large hypothesis spaces, the algorithm is still of interest because it provides a standard against which we may judge the performance of other concept learning algorithms.

In order specify a learning problem for the BRUTE-FORCEMAP LEARNING algorithm we must specify what values are to be used for *P(h)* and for *P(D1h)* (as we shall see, *P (D)* will be determined once we choose the other two). We may choose the probability distributions *P(h)* and *P(D1h)* in any way we wish, to describe our prior knowledge about the learning task. Here let us choose them to be consistent with the following assumptions:

The training data *D* is noise free (i.e., *di = c(xi))* .

The target concept *c* is contained in the hypothesis space H

We have no a priori reason to believe that any hypothesis is more probable than any other.

Given these assumptions, what values should we specify for *P(h)?* Given no prior knowledge that one hypothesis is more likely than another, it is reasonable to assign the same prior probability to every hypothesis *h* in *H* . Furthermore, because we assume the target concept is contained in H we should require that these prior probabilities sum to 1. Together these constraints imply that we should choose What choice shall we make for *P(Dlh)? P(D1h)* is the probability of ob-serving the target values D = *(dl . . . dm)*for the fixed set of instances *(X I . . . x,),* given a world in which hypothesis *h* holds (i.e., given a world in which *h* is the correct description of the target concept c). Since we assume noise-free training data, the probability of observing classification *di* given *h* is just *1* if *di = h(xi)* and 0 if *di # h(xi)* . Therefore, In other words, the probability of data *D* given hypothesis *h* is 1 if *D* is consistent with *h,* and 0 otherwise.

Given these choices for *P(h)* and for *P(Dlh)* we now have a fully-defined problem for the above BRUTE-FORCEMAP LEARNING algorithm. Let us consider the first step of this algorithm, which uses Bayes theorem to compute the posterior probability *P(h1D)* of each hypothesis *h* given the observed training data *D* .

Given these choices for *P(h)* and for *P(Dlh)* we now have a fully-defined problem for the above BRUTE-FORCEMAP LEARNING algorithm. Let us consider the first step of this algorithm, which uses Bayes theorem to compute the posterior probability *P(h1D)* of each hypothesis *h* given the observed training data *D* .Recalling Bayes theorem, we have First consider the case where *h* is inconsistent with the training data *D.* defines *P (D) h)*to be *0* when *h* is inconsistent with *D,* we have

P (~ (D=) -'P(h)- o if *h* is inconsistent with *D P(D)*

The posterior probability of a hypothesis inconsistent with *D* is zero.

MAP Hypotheses and Consistent Learners

The above analysis shows that in the given setting, every hypothesis consistent with *D* is a MAP hypothesis. This statement translates directly into an interesting statement about a general class of learners that we might call consistent learners. We will say that a learning algorithm is a consistent *learner* provided it outputs a hypothesis that commits zero errors over the training examples. Given the above analysis, we can conclude that *every* consistent learner outputs a MAP hypothesis, i f we assume a uniform prior probability distribution over H (i.e., P(hi) = P(hj) for all i, j), and ifwe assume deterministic, noise free training data (i.e., P(D Ih) = 1 i f D and h are consistent, and 0 otherwise).

BAYESIAN BELIEF NETWORKS

As discussed in the previous two sections, the naive Bayes classifier makes significant use of the assumption that the values of the attributes *a1 . . .a*, are condition-ally independent given the target value v. This assumption dramatically reduces the complexity of learning the target function. When it is met, the naive Bayes classifier outputs the optimal Bayes classification. However, in many cases this conditional independence assumption is clearly overly restrictive.

A Bayesian belief network describes the probability distribution governing a set of variables by specifying a set of conditional independence assumptions along with a set of conditional probabilities. In contrast to the naive Bayes classifier, which assumes that *all* the variables are conditionally independent given the value of the target variable, Bayesian belief networks allow stating conditional independence assumptions that apply to *subsets* of the variables. Thus, Bayesian belief networks provide an intermediate approach that is less constraining than the global assumption of conditional independence made by the naive Bayes classifier, but more tractable than avoiding conditional independence assumptions altogether. Bayesian belief networks are an active focus of current research, and a variety of algorithms have been proposed for learning them and for using them for inference.

Representation

A Bayesian belief network (Bayesian network for short) represents the joint prob-ability distribution for a set of variables. For example, the Bayesian network represents the joint probability distribution over the Boolean variables Storm, Lightning, Thunder, Forest Fire, Compare, and *BusTourGroup.* In general, a Bayesian network represents the joint probability

distribution by specifying a set of conditional independence assumptions (represented by a directed acyclic graph), together with sets of local conditional probabilities. Each variable in the joint space is represented by a node in the Bayesian network. For each variable two types of information are specified. First, the network arcs represent the assertion that the variable is conditionally independent of its non descendants in the network given its immediate predecessors in the network. We say Xjis a descendant of
Y if there is a directed path from *Y* to X. Second, a conditional probability table is given for each variable, describing the probability distribution for that variable given the values of its immediate predecessors. The joint probability for any de-sired assignment of values *(y l ,. . . , y,)* to the tuple of network variables *(YI. . . Y,) where Parents (Yi)* denotes the set of immediate predecessors of *Yi* in the net-work. Note the values of *P(yiJParents(Yi))* are precisely the values stored in the conditional probability table associated with node *Yi.*

To illustrate, the Bayesian network represents the joint prob-ability distribution over the Boolean variables Storm, Lightning, Thunder, and Forest-Fire, Campfire, and BusTourGroup. Consider the node *Campfire.* The network nodes and arcs represent the assertion that *Campfire* is conditionally independent of its no descendants *Lightning* and *Thunder,* given its immediate parents *Storm* and *BusTourGroup.* This means that once we know the value of the variables *Storm* and *BusTourGroup,* the variables *Lightning* and *Thunder* provide no additional information about *Campfire.* The right side of the figure shows the conditional probability table associated with the variable *Campfire.* The top left entry in this table, for example, expresses the assertion that
P(Campfire = TruelStorm = True,BusTourGroup = True) = 0.4

Note this table provides only the conditional probabilities of *Campjire* given its parent variables *Storm* and *BusTourGroup.* The set of local conditional probability tables for all the variables, together with the set of conditional independence assumptions described by the network, describe the full joint probability distribution for the network.

One attractive feature of Bayesian belief networks is that they allow a convenient way to represent causal knowledge such as the fact that *Lightning* causes *Thunder.* In the terminology of conditional independence, we express this by stat-ing that *Thunder* is conditionally independent of other variables in the network, given the value of *Lightning.*

Inference

We might wish to use a Bayesian network to infer the value of some target variable (e.g., *ForestFire)* given the observed values of the other variables. Of course, given that we are dealing with random variables it will not generally be correct to assign the target variable a single determined value. What we really wish to infer is the probability distribution for the target variable, which specifies the probability that it will take on each of its possible values given the observed values of the other variables. This inference step can be straightforward if values for all of the other variables in the network are known exactly. In the more general case we may wish to infer the probability distribution for some variable (e.g., *ForestFire)* given observed values for only a subset of the other variables (e.g., *Thunder* and *BusTourGroup* may be the only observed values available). In general, a Bayesian network can be used to compute the probability distribution for any subset of network variables given the values or distributions for any subset of the remaining variables.Exact inference of probabilities in general for an arbitrary Bayesian net-work is known to be NP-hard (Cooper 1990). Numerous methods have

been proposed for probabilistic inference in Bayesian networks, including exact inference methods and approximate inference methods that sacrifice precision to gain efficiency.

Learning Bayesian Belief Networks

Can we devise effective algorithms for learning Bayesian belief networks from training data? This question is a focus of much current research. Several different settings for this learning problem can be considered. First, the network structure might be given in advance, or it might have to be inferred from the training data. Second, all the network variables might be directly observable in each training example, or some might be unobservable.

In the case where the network structure is given in advance and the variables are fully observable in the training examples, learning the conditional probability tables is straightforward. We simply estimate the conditional probability table entries just as we would for a naive Bayes classifier.

In the case where the network structure is given but only some of the variable values are observable in the training data, the learning problem is more difficult. This problem is somewhat analogous to learning the weights for the hidden units in an artificial neural network, where the input and output node values are given but the hidden unit values are left unspecified by the training examples. In fact, Russell et al. (1995) propose a similar gradient ascent procedure that learns the entries in the conditional probability tables. This gradient ascent procedure searches through a space of hypotheses that corresponds to the set of all possible entries for the conditional probability tables. The objective function that is maximized during gradient ascent is the probability P(D1h) of the observed training data D given the hypothesis h. By definition, this corresponds to searching for the maximum likelihood hypothesis for the table entries.

Gradient Ascent Training of Bayesian Networks

The gradient ascent rule given by Russell et al. (1995) maximizes P(D1h) by following the gradient of In P(D Ih) with respect to the parameters that define the conditional probability tables of the Bayesian network. Let *wi;k* denote a single entry in one of the conditional probability tables. In particular, let *wijk* denote the conditional probability that the network variable *Yi* will take on the value *yi,* given that its immediate parents *Ui* take on the values given by *uik.* For example, if *wijk* is the top right entry in the conditional probability table in Figure 6.3, then *Yi* is the variable *Campjire, Ui* is the tuple of its parents *(Stomz,BusTourGroup), yij = True,* and *uik = (False,False).* For example, to calculate the derivative of In *P(D1h)* with respect to the upper-rightmost entry in the table of Figure 6.3 we will have to calculate the quan-tity *P(Campf ire = True , Storm = False, BusTourGroup = Falseld)* for each training example *d* in *D* . When these variables are unobservable for the training example *d* , this required probability can be calculated from the observed variables in *d* using standard Bayesian network inference. In fact, these required quantities are easily derived from the calculations performed during most Bayesian network inference, so learning can be performed at little additional cost whenever the Bayesian network is used for inference and new evidence is subsequently obtained.

Below we derive Equation *(6.25)* following Russell et al. *(1995).* The remainder of this section may be skipped on a first reading without loss of continuity. To simplify notation, in this derivation we will write the abbreviation *Ph(D)* to represent *P (D J h)* .Thus, our problem is to derive the gradient defined by the set of derivatives for all *i* , j, and *k* . Assuming the training examples *d* in the data set D are drawn independently, we write this

derivative as,This last step makes use of the general equality 9= 1f(~)-ax. W can now introduce the values of the variables *Yi* and *Ui = Parents(Yi),*by summing over their possible values *yijl* and *uiu.*

This last step follows from the product rule of probability, Table 6.1. Now consider the rightmost sum in the final expression above. Given that *W i j k = Ph(yijl~ik),*the only term in this sum for which & is nonzero is the term for which j' = j and *i' = i* . Therefore

Applying Bayes theorem to rewrite *Ph(dlyij ,uik)* ,we have

There is one more item that must be considered before we can state the gradient ascent training procedure. In particular, we require that as the weights *wijk* are updated they must remain valid probabilities in the interval [0,1]. We also require that the sum *xjwijk* remains 1 for all *i* , k. These constraints can be satisfied by updating weights in a two-step process. First we update each *wijk*by gradient ascent where *q* is a small constant called the learning rate. Second, we renormalize the weights *wijk* to assure that the above constraints are satisfied. As discussed by Russell et al., this process will converge to a locally maximum likelihood hypothesis for the conditional probabilities in the Bayesian network.

As in other gradient-based approaches, this algorithm is guaranteed only to find some local optimum solution. An alternative to gradient ascent is the EM algorithm discussed which also finds locally maximum likelihood solutions.

Learning the Structure of Bayesian Networks

Learning Bayesian networks when the network structure is not known in advance is also difficult. Cooper and Herskovits (1992) present a Bayesian scoring metric for choosing among alternative networks. They also present a heuristic search algorithm called K2 for learning network structure when the data is fully observ-able. Like most algorithms for learning the structure of

Bayesian networks, K2 performs a greedy search that trades off network complexity for accuracy over the training data. In one experiment K2 was given a set of 3,000 training examples generated at random from a manually constructed Bayesian network containing 37 nodes and 46 arcs. This particular network described potential anesthesia prob-lems in a hospital operating room. In addition to the data, the program was also given an initial ordering over the 37 variables that was consistent with the partial ordering of variable dependencies in the actual network. The program succeeded in reconstructing the correct Bayesian network structure almost exactly, with the exception of one incorrectly deleted arc and one incorrectly added arc.

Constraint-based approaches to learning Bayesian network structure have also been developed (e.g., Spirtes et al. 1993). These approaches infer independence and dependence relationships from the data, and then use these relationships to construct Bayesian networks. Surveys of current approaches to learning Bayesian networks are provided by Heckerman (1995) and Buntine (1994).

THE EM ALGORITHM

In many practical learning settings, only a subset of the relevant instance features might be observable. For example, in training or using the Bayesian belief network of Figure 6.3, we might have data where only a subset of the network variables *Storm, Lightning, Thunder, ForestFire, Campfire,* and *BusTourGroup* have been observed. Many approaches have been proposed to handle the problem of learning in the presence of unobserved variables. As we saw in Chapter 3, if some variable is sometimes observed and sometimes not, then we can use the cases for which it has been observed to learn to predict its values when it is not. In this section we describe the EM algorithm (Dempster

et al. 1977), a widely used approach to learning in the presence of unobserved variables. The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known. The EM algorithm has been used to train Bayesian belief networks (see Heckerman 1995) as well as radial basis function networks discussed ,The EM algorithm is also the basis for many unsupervised clustering algorithms (e.g., Cheeseman et al. 1988), and it is the basis for the widely used Baum-Welch forward-backward algorithm for learning Partially Observable Markov Models (Rabiner 1989).

Estimating Means of k Gaussians

The easiest way to introduce the EM algorithm is via an example. Consider a problem in which the data D is a set of instances generated by a probability distribution that is a mixture of k distinct Normal distributions. This problem setting is illustrated for the case where k = 2 and where the instances are the points shown along the *x* axis. Each instance is generated using a two-step process. First, one of the k Normal distributions is selected at random. Second, a single random instance *xi* is generated according to this selected distribution. This process is repeated to generate a set of data points as shown in the figure. To simplify our discussion, we consider the special case where the selection of the single Normal distribution at each step is based on choosing each with uniform probability, where each of the k Normal distributions has the same variance a2, and where a2 is known. The learning task is to output a hypothesis h = (FI, . . . *pk)* that describes the means of each of the k distributions.

A second property, the variance, captures the "width or "spread" of the probability distribution; that is, it captures how far the random variable is expected to vary from its mean value.

The variance describes the expected squared error in using a single observation of Y to estimate its mean $E[Y]$ .The square root of the variance is called the *standard deviation* of Y , denoted oy .

UNIT-IV

EVOLUTIONARY LEARNING

Genetic Algorithms, genetic operators; Genetic programming; Ensemble learning: Boosting, bagging; Dimensionality reduction: Linear discriminate analysis, principal component analysis (JAX-RPC).

A collection of hypotheses called the current population is updated by replacing some fraction of the population by offspring of the most fit current hypotheses. The process forms a generate-and-test beam-search of hypotheses, in which vari-ants of the best current hypotheses are most likely to be considered next. The popularity of GAS is motivated by a number of factors including:

- Evolution is known to be a successful, robust method for adaptation within biological systems.
- GAS can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model. Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

We also describe a variant called genetic programming, in which entire computer programs are evolved to certain fitness criteria. Genetic algorithms and genetic programming are two of the more popular approaches in a field that is sometimes called evolutionary computation. In the final section we touch on selected topics in the study of biological evolution, including the Baldwin effect, which describes an interesting interplay between the learning capabilities of single individuals and the rate of evolution of the entire population.

GENETIC ALGORITHMS

The problem addressed by GAS is to search a space of candidate hypotheses to identify the best hypothesis. In GAS the "best hypothesis" is defined as the one that optimizes a predefined numerical measure for the problem at hand, called the hypothesis *Jitness*. For example, if the learning task is the problem of approximating an unknown function given training examples of its input and output, then fitness could be defined as the accuracy of the hypothesis over this training data. If the task is to learn a strategy for playing chess, fitness could be defined as the number of games won by the individual when playing against other individuals in the current population.

Although different implementations of genetic algorithms vary in their details, they typically share the following structure: The algorithm operates by iteratively updating a pool of hypotheses, called the population. On each iteration, all members of the population are evaluated according to the fitness function. A new population is then generated by probabilistically selecting the fit individuals from the current population. Some of these selected individuals are carried forward into the next generation population intact. Others are used as the basis for creating new offspring individuals by applying genetic operations such as crossover and mutation.

Fitness: A function that assigns an evaluation score, given a hypothesis.

Fitnessdhreshold: A threshold specifying the termination criterion.

The number of hypotheses to be included in the population.

The fraction of the population to be replaced by Crossover at each step.

The mutation rate.

Initialize population: P c Generate p hypotheses at random

Evaluate: For each h in P , compute Fitness(h)' While [max Fitness(h)]< Fitnessdhreshold do h

Select: F'robabilistically select (1 - *r) p* members of *P* to add to *Ps.* The probability Pr(hi) of selecting hypothesis *hi* from *P* is given by

2. Crossover: Probabilistically select pairs of hypotheses from *P* , according to *&(hi)* given above. For each pair, *(h l ,h2),*produce two offspring by applying the Crossover operator. Add all offspring to *P,*.

Mutate: Choose *m* percent of the members of *P,* with uniform probability. For each, invert one randomly selected bit in its representation.

Update: P t *P,*.

5. Evaluate: for each *h* in *P ,* compute *Fitness(h)*

Return the hypothesis from *P* that has the highest fitness.

A prototypical genetic algorithm. A population containing *p* hypotheses is maintained. On each iteration, the successor population *Ps* is formed by probabilistically selecting current hypotheses according to their fitness and by adding new hypotheses. New hypotheses are created by applying a crossover operator to pairs of most fit hypotheses and by creating single point mutations in the resulting generation of hypotheses. This process is iterated until sufficiently fit hypotheses are discovered. Typical crossover and mutation operators are defined in a subsequent table.

The inputs to this algorithm include the fitness function for ranking candidate hypotheses, a threshold defining an acceptable level of fitness for terminating the algorithm, the size of the population to be maintained, and parameters that determine how successor populations are to be generated: the fraction of the population to be replaced at each generation and the mutation rate.

Notice in this algorithm each iteration through the main loop produces a new generation of hypotheses based on the current population. First, a certain number of hypotheses from the current population are selected for inclusion in the next generation. Thus, the probability that a hypothesis will be selected is

proportional to its own fitness and is inversely proportional to the fitness of the other competing hypotheses in the current population.

Once these members of the current generation have been selected for inclusion in the next generation population, additional members are generated using a crossover operation. Crossover, defined in detail in the next section, takes two parent hypotheses from the current generation and creates two offspring hypotheses by recombining portions of both parents. The parent hypotheses are chosen probabilistically from the current population, again using the probability function given, after new members have been created by this crossover operation, the new generation population now contains the desired number of members. At this point, a certain fraction m of these members are chosen at random and random mutations all performed to alter these members? This GA algorithm thus performs a randomized, parallel beam search for hypotheses that perform well according to the fitness function. In the follow-ing subsections, we describe in more detail the representation of hypotheses and genetic operators used in this algorithm.

Representing Hypotheses

Hypotheses in GAS are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover. The hypotheses represented by these bit strings can be quite complex. For example, sets of if-then rules can easily be represented in this way, by choosing an encoding of rules that allocates specific substrings for each rule precondition and post condition. To see how if-then rules can be encoded by bit strings, first consider how we might use a bit string to describe a constraint on the value of a single attribute.

To pick an example, consider the attribute Outlook, which can take on any of the three values Sunny, Overcast, or Rain. One obvious way to represent a constraint on Outlook is to use a bit string of length three, in which each bit position corresponds to one of its three possible values. Placing a 1 in some position indicates that the attribute is allowed to take on the corresponding value. For example, the string 010 represents the constraint that *Outlook* must take on the second of these values, , or *Outlook = Overcast.* Similarly, the string 011 represents the more general constraint that allows two possible values, or *(Outlook = Overcast v Rain).* Note 111 represents the most general possible constraint, indicating that we don't care which of its possible values the attribute takes on

Given this method for representing constraints on a single attribute, con-junctions of constraints on multiple attributes can easily be represented by concatenating the corresponding bit strings. For example, consider a second attribute, *Wind,* that can take on the value *Strong* or *Weak. A* rule precondition such as *(Outlook = Overcast* V *Rain) A (Wind = Strong)* can then be represented by the following bit string of length five

Rule post conditions (such as *PlayTennis = yes)* can be represented in a similar fashion. Thus, an entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule post condition. For example, the rule would be represented by the string where the first three bits describe the "don’t care" constraint on *Outlook,* the next where the first three bits describe the "don't care" constraint on *Outlook,* the next two bits describe the constraint on *Wind ,* and the final two bits describe the rule post condition (here we assume *Play Tennis* can take on the values *Ye s* or *No)* . Note the bit string representing the rule contains a substring for each attribute in the hypothesis space, even if that attribute is not

constrained by the rule pre-conditions. This yields a fixed length bit-string representation for rules, in which substrings at specific locations describe constraints on specific attributes. Given this representation for single rules, we can represent sets of rules by similarly concatenating the bit string representations of the individual rules.

In designing a bit string encoding for some hypothesis space, it is useful to arrange for every syntactically legal bit string to represent a well-defined hypothesis. To illustrate, note in the rule encoding in the above paragraph the bit string 111 10 11 represents a rule whose post condition does not constrain the target attribute *PlayTennis.* If we wish to avoid considering this hypothesis, we may employ a different encoding (e.g., allocate just one bit to the *PlayTennis* post-condition to indicate whether the value is *Ye s* or *No),* alter the genetic operators so that they explicitly avoid constructing such bit strings, or simply assign a very low fitness to such bit strings.

Genetic Operators

The generation of successors in a GA is determined by a set of operators that recombine and mutate selected members of the current population.

These operators correspond to idealized versions of the genetic operations found in biological evolution. The crossover operator produces two new offspring from two parent strings, by copying selected bits from each parent. The bit at position i in each offspring is copied from the bit at position i in one of the two parents. The choice of which parent contributes the bit for position i is determined by an additional string called the *crossover mask.* Consider the topmost of the two offspring in this case. This offspring takes its first five bits from the first parent and its remaining six bits from the second parent, because the crossover mask 11111000000 specifies these choices for each of the bit

positions. The second offspring uses the same crossover mask, but switches the roles of the two parents. Therefore, it contains the bits that were not used by the first offspring. In single-point crossover, the crossover mask is always constructed so that it begins with a string containing *n* contiguous Is, followed by the necessary number of 0s to complete the string. This results in offspring in which the first n bits are contributed by one parent and the remaining bits by the second parent. Each time the single-point crossover operator is applied, the crossover point *n* is chosen at random, and the crossover mask is then created and applied.

In *two-point crossover,* offspring are created by substituting intermediate segments of one parent into the middle of the second parent string. Put another way, the crossover mask is a string beginning with *no* zeros, followed by a contiguous string of *nl* ones, followed by the necessary number of zeros to complete the string. Each time the two-point crossover operator is applied, a mask is generated by randomly choosing the integers *no* and *nl.* For instance, in the example shown in Table 9.2 the offspring are created using a mask for which *no* = 2 and *n* 1 = 5. Again, the two offspring are created by switching the roles played by the two parents.

Uniform crossover combines bits sampled uniformly from the two parents, In this case the crossover mask is generated as a random bit string with each bit chosen at random and independent of the others.In addition to recombination operators that produce offspring by combining parts of two parents, a second type of operator produces offspring from a single parent. In particular, the *mutation* operator produces small random changes to the bit string by choosing a single bit at random, then changing its value.

Some GA systems employ additional operators, especially operators that are

specialized to the particular hypothesis representation used by the system. For example, Grefenstette et al. (1991) describe a system that learns sets of rules for robot control. It uses mutation and crossover, together with an operator for specializing rules. Janikow (1993) describes a system that learns sets of rules using operators that generalize and specialize rules in a variety of directed ways (e.g., by explicitly replacing the condition on an attribute by "don't care").

Fitness Function and Selection

The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population. If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples. Often other criteria may be included as well, such as the complexity or generality of the rule. More generally, when the bit-string hypothesis is interpreted as a complex procedure (e.g., when the bit string represents a collection of if-then rules that will be chained together to control a robotic device), the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.

The probability that a hypothesis will be selected is given by the ratio of its fitness to the fitness of other members of the current population as seen ,This method is sometimes called jitness proportionate selection, or roulette wheel selection. Other methods for using fitness to select hypotheses have also been proposed. For example, in containing a small number of defined bits (i.e., containing a large number of *'s), and especially when these defined bits are near one another within the bit string.

The schema theorem is perhaps the most widely cited characterization of population evolution within a GA. One way in which it is incomplete is that it fails to consider the (presumably) positive effects of crossover and mutation. Numerous more recent theoretical analyses have been proposed, including analyses based on Markov chain models and on statistical mechanics models. See, for example, Whitley and Vose (1995) and Mitchell (1996).

GENETIC PROGRAMMING

Genetic programming (GP) is a form of evolutionary computation in which the individuals in the evolving population are computer programs rather than bit strings. Koza (1992) describes the basic genetic programming approach and presents a broad range of simple programs that can be successfully learned by GP.

Representing Programs

Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program. Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes. For example, illustrates this tree representation for the function sin(x) + *J*-. To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., sin, cos, *J*,+, -, ex-ponential~),as well as the terminals (e.g., x, *y* , constants such as 2). The genetic programming algorithm then uses an evolutionary search to explore the vast space of programs that can be described using these primitives. As in a genetic algorithm, the prototypical genetic programming algorithm maintains a population of individuals (in this case, program trees). On each iteration, it produces a new generation of individuals using selection, crossover, and

mutation. The fitness of a given individual program in the population is typically determined by executing the program on a set of training data. Crossover operations are performed by replacing a randomly chosen sub tree of one parent program by a sub tree from the other parent program. Koza (1992) describes a set of experiments applying a GP to a number of applications. In his experiments, 10% of the current population, selected probabilistically according to fitness is retained unchanged in the next generation. The remainder of the new generation is created by applying crossover to pairs of programs from the current generation, again selected probabilistically accord-ing to their fitness. The mutation operator was not used in this particular set of experiments.

Remarks on Genetic Programming

Genetic programming extends genetic algorithms to the evolution of complete computer programs. Despite the huge size of the hypothesis space it must search, genetic programming has been demonstrated to produce intriguing results in a number of applications. A comparison of GP to other methods for searching through the space of computer programs, such as hill climbing and simulated annealing, is given by O'Reilly and Oppacher (1994).

The primitive functions used by the GP to construct its programs are functions that edit the seed circuit by inserting or deleting circuit components and wiring connections. The fitness of each program is calculated by simulating the circuit it outputs (using the SPICE circuit simulator) to de-termine how closely this circuit meets the design specifications for the desired filter. More precisely, the fitness score is the sum of the magnitudes of errors between the desired and actual circuit output at 101 different input frequencies. In this case, a population of size 640,000 was maintained, with selection producing

10% of the successor population, crossover producing 89%, and mutation producing 1%. The system was executed on a 64-node parallel processor. Within the first randomly generated population, the circuits produced were so unreasonable that the SPICE simulator could not even simulate the behavior of 98% of the circuits. The percentage of unsimulatable circuits dropped to 84.9% following the first generation, to 75.0% following the second generation, and to an average of 9.6% over succeeding generations. The fitness score of the best circuit in the initial population was 159, compared to a score of 39 after 20 generations and a score of 0.8 after 137 generations. The best circuit, pro-duced after 137 generations, exhibited performance very similar to the desired behavior.

In most cases, the performance of genetic programming depends crucially on the choice of representation and on the choice of fitness function. For this reason, an active area of current research is aimed at the automatic discovery and incorporation of subroutines that improve on the original set of primitive functions, thereby allowing the system to dynamically alter the primitives from which it constructs individuals. See, for example, Koza (1994).

MODELS OF EVOLUTION AND LEARNING

In many natural systems, individual organisms learn to adapt significantly during their lifetime. At the same time, biological and social processes allow their species to adapt over a time frame of many generations. One interesting question regarding evolutionary systems is "What is the relationship between learning during the lifetime of a single individual, and the longer time frame species-level learning afforded by evolution?'

Lamarckian Evolution

Larnarck was a scientist who, in the late nineteenth century, proposed that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime. In particular, he proposed that experiences of a single organism directly affected the genetic makeup of their offspring: If an individual learned during its lifetime to avoid some toxic food, it could pass this trait on genetically to its offspring, which therefore would not need to learn the trait. This is an attractive conjecture, because it would presumably allow for more efficient evolutionary progress than a generate-and-test process (like that of GAS and GPs) that ignores the experience gained during an individual's lifetime. Despite the attractiveness of this theory, current scientific evidence over whelmingly contradicts Lamarck's model.

Baldwin Effect

Although Lamarckian evolution is not an accepted model of biological evolution, other mechanisms have been suggested by which individual learning can alter the course of evolution. One such mechanism is called the Baldwin effect, after J. M. Baldwin (1896), who first suggested the idea. The Baldwin effect is based on the following observations:

If a species is evolving in a changing environment, there will be evolution-ary pressure to favor individuals with the capability to learn during their lifetime. For example, if a new predator appears in the environment, then individuals capable of learning to avoid the predator will be more successful than individuals who cannot learn. In effect, the ability to learn allows an individual to perform a small local search during its lifetime to maximize its fitness. In contrast, nonearning individuals whose fitness is fully determined by their genetic makeup will operate at a relative disadvantage.

Those individuals who are able to learn many traits will rely less strongly on their genetic code to "hard-wire" traits. As a result, these individuals can support a more diverse gene pool, relying on individual learning to overcome the "missing" or "not quite optimized" traits in the genetic code. This more diverse gene pool can, in turn, support more rapid evolutionary adaptation. Thus, the ability of individuals to learn can have an indirect accelerating effect on the rate of evolutionary adaptation for the entire population.

To illustrate, imagine some new change in the environment of some species, such as a new predator. Such a change will selectively favor individuals' capable of learning to avoid the predator. As the proportion of such self-improving individuals in the population grows, the population will be able to support a more diverse gene pool, allowing evolutionary processes (even non-Lamarckian generate-and-test processes) to adapt more rapidly. This accelerated adaptation may in turn enable standard evolutionary processes to more quickly evolve a genetic (non learned) trait to avoid the predator (e.g., an instinctive fear of this animal). Thus, the Baldwin effect provides an indirect mechanism for individual learning to positively impact the rate of evolutionary progress. By increasing survivability and genetic diversity of the species, individual learning sup-ports more rapid evolutionary progress, thereby increasing the chance that the species will evolve genetic, non learned traits that better fit the new environment.

There have been several attempts to develop computational models to study the Baldwin effect. For example, Hinton and Nowlan (1987) experimented with evolving a population of simple neural networks, in which some network weights were fixed during the individual network "lifetime," while others were trainable. The genetic makeup of the individual determined which weights were train-able and which were fixed. In their experiments, when no

individual learning was allowed, the population failed to improve its fitness over time. However, when individual learning was allowed, the population quickly improved its fit-ness. During early generations of evolution the population contained a greater proportion of individuals with many trainable weights. However, as evolution proceeded, the number of fixed, correct network weights tended to increase, as the population evolved toward genetically given weight values and toward less dependence on individual learning of weights. Additional computational studies of the Baldwin effect have been reported by Belew (1990), Harvey (1993), and French and Messinger (1994). An excellent overview of this topic can be found in Mitchell (1996). A special issue of the journal *Evolutionary Computa-tion* on this topic (Turney et al. 1997) contains several articles on the Baldwin effect.

PARALLELIZING GENETIC ALGORITHMS

GAS is naturally suited to parallel implementation, and a number of approaches to parallelization have been explored. *Coarse grain* approaches to parallelization subdivide the population into somewhat distinct groups of individuals, called *demes.* Each deem is assigned to a different computational node, and a standard GA search is performed at each node. Communication and cross-fertilization between demes occurs on a less frequent basis than within demes. Transfer between demes occurs by a *migration* process, in which individuals from one deme are copied or transferred to other demes. This process is modeled after the kind of cross-fertilization that might occur between physically separated subpopulations of biological species. One benefit of such approaches is that it reduces the crowd-ing problem often encountered in nonparallel GAS, in which the system falls into a local optimum due to the early appearance of a genotype that comes to dominate the entire population. Examples of coarse-grained parallel GAS are described by

Tanese (1989) and by Cohoon et al. (1987).In contrast to coarse-grained parallel implementations of GAS, fine grained implementations typically assign one processor per individual in the population.

Recombination then takes place among neighboring individuals. Several different types of neighborhoods have been proposed, ranging from planar grid totorus. Examples of such systems are described by Spiessens and Manderick(1991). An edited collection of papers on parallel GAS is available in Stender(1993).

SUMMARY AND FURTHER READING

The main points of this chapter include:

Genetic algorithms (GAS) conduct a randomized, parallel, hill-climbing search for hypotheses that optimize a predefined fitness function. The search performed by GAS is based on an analogy to biological evolution. A diverse population of competing hypotheses is maintained. At each one of the most expressive and human readable representations for learned hypotheses is sets of if-then rules. This chapter explores several algorithms for learning such sets of rules. One important special case involves learning sets of rules containing variables, called first-order Horn clauses. Because sets of first-order Horn clauses can be interpreted as programs in the logic programming language PROLOG, learning them is often called inductive logic programming (ILP). This chapter examines several approaches to learning sets of rules, including an approach based on inverting the deductive operators of mechanical theorem proves.

INTRODUCTION

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

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

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

capable of learning such rule sets may be viewed as an algorithm for automatically inferring PROLOG programs from examples. In this chapter we explore learning algorithms capable of learning such rules, given appropriate sets of training examples.

In practice, learning systems based on first-order representations have been successfully applied to problems such as learning which chemical bonds fragment in a mass spectrometer (Buchanan 1976; Lindsay 1980), learning which chemical substructures produce mutagenic activity (a property related to carcinogenicity) (Srinivasan et al. 1994), and learning to design finite element meshes to analyze stresses in physical structures (Dolsak and Muggleton 1992).

In each of these applications, the hypotheses that must be represented involve relational assertions that can be conveniently expressed using first-order representations, while they are very difficult to describe using propositional representations.

To elaborate, imagine we have a subroutine LEARN-ONE-RULE that accepts a set of positive and negative training examples as input, then outputs a single rule that covers many of the positive examples and few of the negative examples. We require that this is input rule have high accuracy, but not necessarily high coverage.

In this section we consider learning only propositional rules. In later sections, we extend these algorithms to learn first-order Horn clauses.

SEQUENTIAL COVERING ALGORITHMS

Here we consider a family of algorithms for learning rule sets based on the strategy of learning one rule, removing the data it covers, then iterating this process. Such algorithms are called sequential covering algorithms. To elaborate, imagine we have a subroutine LEARN-ONE-RULE that accepts a set of positive and negative training examples as input, then outputs a single rule that covers many of the positive examples and few of the negative examples. We require that this is input rule have high accuracy, but not necessarily high coverage. By high accuracy, we mean the predictions it makes should be correct. By accepting low coverage, we mean it need not make predictions for every training example.

Given this LEARN-ONE-RULE subroutine for learning a single rule, one obvious approach to learning a set of rules is to invoke LEARN-ONE-RULE on all the available training examples, remove any positive examples covered by the rule it learns, then invoke it again to learn a second rule based on the remaining train-ing examples. This procedure can be iterated as many times as desired to learn a disjunctive set of rules that together cover any desired fraction of the positive examples. This is called a sequential covering algorithm because it sequentially learns a set of rules that together cover the full set of positive examples. The final set of rules can then be sorted so that more accurate rules will be considered first when a new instance must be classified. This sequential covering algorithm is one of the most widespread approaches to learning disjunctive sets of rules. It reduces the problem of learning a disjunctive set of rules to a sequence of simpler problems, each requiring that a single conjunctive rule be learned. Because it performs a greedy search, formulating a sequence of rules without backtracking, it is not

guaranteed to find the smallest or best set of rules that cover the training examples.

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

General to Specific Beam Search

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

This approach to implementing LEARN-ONE-RULE performs a general-to-specific search through the space of possible rules in search of a rule with high accuracy, though perhaps incomplete coverage of the data. As in decision tree learning, there are many ways to define a measure to select the "best"

descendant. To follow the lead of ID3 let us for now define the best descendant as the one whose covered examples have the lowest entropy

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

UNIT V

CLUSTERING

Similarity and distance measures, outliers, hierarchical methods, partitional algorithms, clustering large databases, clustering with categorical attributes, comparison.

What is Cluster Analysis?

- Cluster: a collection of data objects
  - Similar to one another within the same cluster
  - Dissimilar to the objects in other clusters
- Cluster analysis - Grouping a set of data objects into clusters
- Clustering is unsupervised classification: no predefined classes
- Typical applications
  - As a stand-alone tool to get insight into data distribution
  - As a preprocessing step for other algorithms

General Applications of Clustering:

- Pattern Recognition
- Spatial Data Analysis
  - create thematic maps in GIS by clustering feature spaces
  - detect spatial clusters and explain them in spatial data mining
- Image Processing
- Economic Science (especially market research)
- WWW
  - Document classification
  - Cluster Weblog data to discover groups of similar access patterns

Examples of Clustering Applications:

-Marketing: Help marketers discover distinct groups in their customer bases, and then use this knowledge to develop targeted marketing programs

-Land use: Identification of areas of similar land use in an earth observation database

-Insurance: Identifying groups of motor insurance policy holders with a high average claim cost

-City-planning: Identifying groups of houses according to their house type, value, and geographical location

-Earth-quake studies: Observed earth quake epicenters should be clustered along continent faults

Ratio-Scaled Variables:

-Ratio-scaled variable: a positive measurement on a nonlinear scale, approximately at exponential scale,such as AeBt or Ae-Bt
-Methods: treat them like interval-scaled variables not a good choice! (why?) apply logarithmic transformation yif = log(xif) treat them as continuous ordinal data treat their rank as interval-scaled.

Variables of Mixed Types:

-A database may contain all the six types of variables
-symmetric binary, asymmetric binary, nominal, ordinal, interval and ratio.

Categorization of Major Clustering Methods:

* Partitioning algorithms: Construct various partitions and then evaluate them by some criterion
* Hierarchy algorithms: Create a hierarchical decomposition of the set of data (or objects) using some criterion
* Density-based: based on connectivity and density functions
* Grid-based: based on a multiple-level granularity structure
* Model-based: A model is hypothesized for each of the clusters and the idea is to findthe best fit of that model to each other

Partitioning Algorithms: Basic Concept

- Partitioning method: Construct a partition of a database D of n objects into a set of k clusters
- Given a k, find a partition of k clusters that optimizes the chosen partitioning criterion
    - Global optimal: exhaustively enumerate all partitions.
    - Heuristic methods: k-means and k-medoids algorithms.
    - K-means (MacQueen‘67): Each cluster is represented by the center of thecluster.
    - k-medoids or PAM (Partition around medoids) (Kaufman & Rousseeuw‘87): Each cluster is represented by one of the objects in the cluster.

The K-Means Clustering Method:

- Given k, the k-means algorithm is implemented in 4 steps:
    - Partition objects into k nonempty subsets
    - Compute seed points as the centroids of the clusters of the current partition. The centroid is the center (mean point) of the cluster.
    - Assign each object to the cluster with the nearest seed point.
    - Go back to Step 2, stop when no more new assignment.

Example

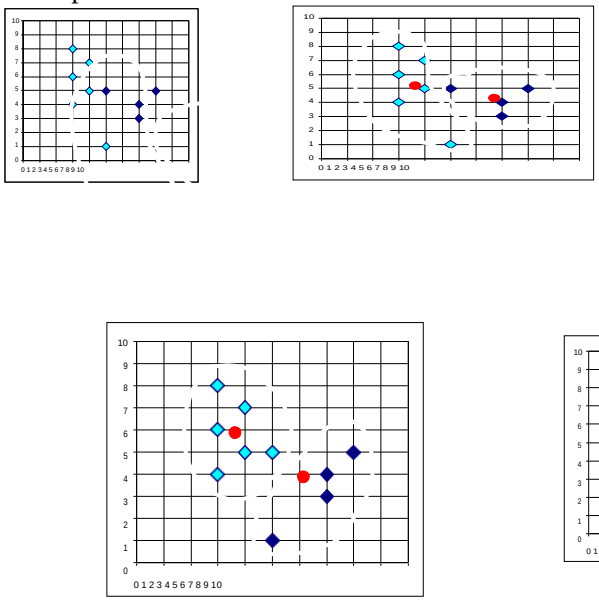


Fig.5.1 Comments on the K-Means Method

– Strength
  - Relatively efficient: O(tkn), where n is # objects, k is # clusters, and t is #iterations. Normally, k, t << n.
  - Often terminates at a local optimum. The global optimum may be found using techniques such as: deterministic annealing and genetic algorithms
- Weakness
  - Applicable only when mean is defined, then what about categorical data?
  - Need to specify k, the number of clusters, in advance
  - Unable to handle noisy data and outliers
  - Not suitable to discover clusters with non-convex shapes

Variations of the K-Means Method

- A few variants of the k-means which differ in
  -Selection of the initial k means
  -Dissimilarity calculations
  -Strategies to calculate cluster means
- Handling categorical data: k-modes (Huang'98)
  -Replacing means of clusters with modes
  -Using new dissimilarity measures to deal with categorical objects
  -Using a frequency-based method to update modes of clusters
  -A mixture of categorical and numerical data: k-prototype method

The K-Medoids Clustering Method

- Find representative objects, called medoids, in clusters
- PAM (Partitioning Around Medoids, 1987)
- starts from an initial set of medoids and iteratively replaces one of the medoids by one of the non-medoids if it improves the total distance of the resultingclustering
- PAM works effectively for small data sets, but does not scale well for large data sets
- CLARA (Kaufmann & Rousseeuw, 1990)
- CLARANS (Ng & Han, 1994): Randomized sampling
- Focusing + spatial data structure (Ester et al., 1995)

Edoids Clustering Method

- Find representative objects, called medoids, in clusters
- PAM (Partitioning Around Medoids, 1987)
- starts from an initial set of medoids and iteratively replaces one of the medoids by one of the non-medoids if it improves the total distance of the resultingclustering
- PAM works effectively for small data sets, but does not scale well for large data sets
- CLARA (Kaufmann & Rousseeuw, 1990)
- CLARANS (Ng & Han, 1994): Randomized sampling
- Focusing + spatial data structure (Ester et al., 1995)

PAM (Partitioning Around Medoids) (1987)

- PAM (Kaufman and Rousseeuw, 1987), built in Splus

- Use real object to represent the cluster
  - Select k representative objects arbitrarily
  - For each pair of non-selected object h and selected object i, calculate the total swapping cost TCih
  - For each pair of i and h,
  - If TCih < 0, i is replaced by h
  - Then assign each non-selected object to the most similar representative object
  - repeat steps 2-3 until there is no change

PAM Clustering: Total swapping cost

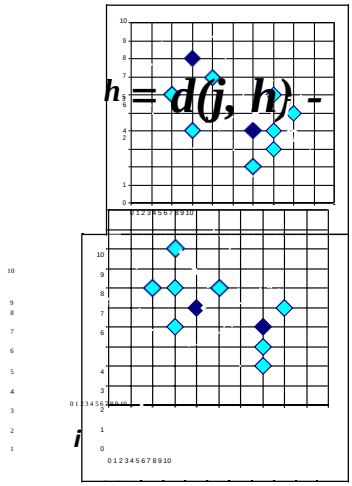


Fig 5.2 PAM Clustering: Total swapping cost

CLARA (Clustering Large Applications) (1990)

- CLARA (Kaufmann and Rousseeuw in 1990)
- Built in statistical analysis packages, such as S+
- It draws multiple samples of the data set, applies PAM on each sample, and gives the best clustering as the output
- Strength: deals with larger data sets than PAM
- Weakness:
- Efficiency depends on the sample size
- A good clustering based on samples will not necessarily represent a good clustering of the whole data set if the sample is biased

CLARANS (―Randomized‖ CLARA) (1994)

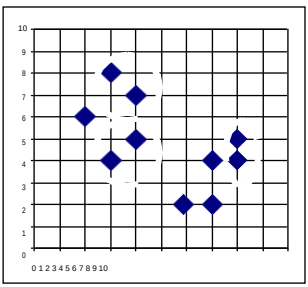
- CLARANS (A Clustering Algorithm based on Randomized Search) (Ng and Han‘94)
- CLARANS draws sample of neighbors dynamically
- The clustering process can be presented as searching a graph where every node isa potential solution, that is, a set of k medoids
- If the local optimum is found, CLARANS starts with new randomly selected node in search for a new local optimum
- It is more efficient and scalable than both PAM and CLARA
- Focusing techniques and spatial access structures may further improve its performance (Ester etal.‘95)

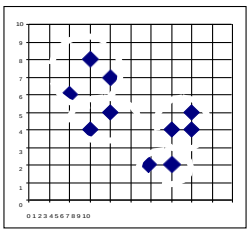
Hierarchical Clustering

Use distance matrix as clustering criteria. This method does not require the number of clusters k as an input, but needs a termination condition

AGNES (Agglomerative Nesting)

- Introduced in Kaufmann and Rousseeuw (1990)
- Implemented in statistical analysis packages, e.g., Splus
- Use the Single-Link method and the dissimilarity matrix.
- Merge nodes that have the least dissimilarity
- Go on in a non-descending fashion
- Eventually all nodes belong to the same cluster





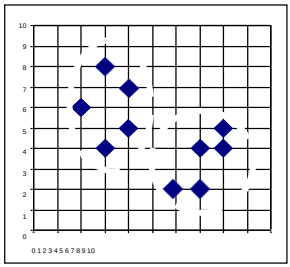


Fig 5.3 Agglomerative Nesting

A Dendrogram Shows How the Clusters are Merged Hierarchically

- Decompose data objects into a several levels of nested partitioning (tree of clusters), called a dendrogram.
- A clustering of the data objects is obtained by cutting the dendrogram at the desired level, then each connected component forms a cluster.

DIANA (Divisive Analysis)

- Introduced in Kaufmann and Rousseeuw (1990)
- Implemented in statistical analysis packages, e.g., Splus
- Inverse order of AGNES
- Eventually each node forms a cluster on its own

More on Hierarchical Clustering Methods

- Major weakness of agglomerative clustering methods
  - do not scale well: time complexity of at least O(n2), where n is the number of total objects
  - can never undo what was done previously
- Integration of hierarchical with distance-basedclustering
- BIRCH (1996): uses CF-tree and incrementally adjusts the quality of sub-clusters
- CURE (1998): selects well-scattered points from the cluster and then shrinks them towards the center of the cluster by a specified fraction
- CHAMELEON (1999): hierarchical clustering using dynamic modeling

BIRCH (1996)

- Birch: Balanced Iterative Reducing and Clustering using Hierarchies,by Zhang, Ramakrishnan, Livny (SIGMOD‘96)
- Incrementally construct a CF (Clustering Feature) tree, a hierarchical data structure for multiphase clustering
- Phase 1: scan DB to build an initial in-memory CF tree (a multi-level compression of the data that tries to preserve the inherent clustering structure of the data)
- Phase 2: use an arbitrary clustering algorithm to cluster the leaf nodes

of the CF- tree

- *Scales linearly*: finds a good clustering with a single scan and improves the quality with a few additional scans
- *Weakness:* handles only numeric data, and sensitive to the order of the data record.

Rock Algorithm and CHAMELEON.

- ROCK: Robust Clustering using linKs,

by S. Guha, R. Rastogi, K. Shim (ICDE'99).

- Use links to measure similarity/proximity
- Not distance based
- Computational complexity:
- Basic ideas:
- Similarity function and neighbors:

Let *T1* = {1,2,3}, *T2*={3,4,5}

Rock: Algorithm

- Links: The number of common neighbours for the two points.

{1,2,3}, {1,2,4}, {1,2,5}, {1,3,4}, {1,3,5}
{1,4,5}, {2,3,4}, {2,3,5}, {2,4,5}, {3,4,5}
{1,2,3} 3 {1,2,4}

- Algorithm
- Draw random sample
- Cluster with links
- Label data in disk

CHAMELEON

- CHAMELEON: hierarchical clustering using dynamic modeling, by G. Karypis, E.H. Han and V. Kumar'99
- Measures the similarity based on a dynamic model

Two clusters are merged only if the *interconnectivity* and *closeness (proximity)* between two clusters are high *relative to* the internal interconnectivity of the clusters and closeness of items within the clusters.

- A two phase algorithm

1. Use a graph partitioning algorithm: cluster objects into a large number of relatively small sub-clusters
2. Use an agglomerative hierarchical clustering algorithm: find the genuine clusters by repeatedly combining these sub-clusters

AGGLOMERATIVE HIERARCHICAL CLUSTERING

Algorithms of hierarchical cluster analysis are divided into the two categories divisible algorithms and agglomerative algorithms. A *divisible algorithm* starts from the entire set of samples X and divides it into a partition of subsets, then divides each subset into smaller sets, and so on. Thus, a divisible algorithm generates a sequence of partitions that is ordered from a coarser one to a finer one. An *agglomerative algorithm* first regards each object as an initial cluster. The clusters are merged into a coarser partition, and the merging process proceeds until the trivial partition is obtained: all objects are in one large cluster. This process of clustering is a bottom-up process, where partitions from a finer one to a coarser one.

Most agglomerative hierarchical clustering algorithms are variants of the *single-link* or *complete-link* algorithms. In the single-link method, the distance between two clusters is the *minimum* of the distances between all pairs of samples drawn from the two clusters (one element from the first cluster, the other from the second). In the complete-link algorithm, the distance between two clusters is the *maximum* of all distances between all pairs drawn from the two clusters. A graphical illustration of these two distance measures is given. The basic steps of the agglomerative clustering algorithm are the same.

Place each sample in its own cluster. Construct the list of inter-cluster distances for all distinct unordered pairs of samples, and sort this list in ascending order. Step through the sorted list of distances, forming for each distinct threshold value d_k a graph of the samples where pairs samples closer than d_k are connected into a new cluster by a graph edge. If all the samples are members of a connected graph, stop. Otherwise, repeat this step.

The output of the algorithm is a nested hierarchy of graphs, which can be cut at the desired dissimilarity level forming a partition (clusters) identified by simple connected components in the corresponding sub graph. Let us consider five points $\{x_1, x_2, x_3, x_4, x_5\}$ with the following coordinates as a two-dimensional sample for clustering:

For this example, we selected two-dimensional points because it is easier to graphically represent these points and to trace all the steps in the clustering algorithm.

The distances between these points using the Euclidian measure are $d(x_1 , x_2) = 2$, $d(x_1, x_3) = 2.5$, $d(x_1, x_4) = 5.39$, $d(x_1, x_5) = 5$

$d(x_1 , x_3) = 1.5$, $d(x_2, x_4) = 5$, $d(x_2, x_5) = 5.29$,

$d(x_3 , x_4) = 3.5$, $d(x_3, x_5) = 4.03$, $d(x_4, x_5) = 2$

The distances between points as clusters in the first iteration are the same for

both single- link and complete-link clustering. Further computation for these two algorithms is different. Using agglomerative single-link clustering, the following steps are performed to create a cluster and to represent the cluster structure as a dendrogram.

Hierarchical and Non-Hierarchical Clustering

There are two main types of clustering techniques, those that create a hierarchy of clusters and those that do not. The hierarchical clustering techniques create a hierarchy of clusters from small - big. The main reason for this is that, as was already stated, clustering is an unsupervised learning technique, and as such, there is no absolutely correct answer. For this reason and depending on the particular application of the clustering, fewer or greater numbers of clusters may be desired. With a hierarchy of clusters defined it is possible to choose the number of clusters that are desired. At the extreme it is possible to have as many clusters as here are records in the database. In thiscase the records within the cluster are optimally similar to each other (since there is only one) and certainly different from the other clusters. But of course such a clustering technique misses the point in the sense that the idea of clustering is to find useful patters in the database that summarize it and make it easier to understand. Any clustering algorithm that ends up with as many clusters as there are records has not helped the user understand the data any better. Thus one of the main points about clustering is that there are many fewer clusters than there are original records. Exactly how many clusters should be formed is a matter of interpretation.

The advantage of hierarchical clustering methods is that they allow the end user to choose from either many clusters or only a few. The hierarchy of clusters is usually viewed as a tree where the smallest clusters merge together to create the next highest level of clusters and those at that level merge together to create the next highest level of clusters. below Figure shows how several clusters might form a hierarchy. When a hierarchy of clusters like this is created the user can determine what the right number of clusters is that adequately summarizes the data while still providing useful information (at the other extreme a single cluster containing all the records is a great summarization but does not contain enough specific information to be useful). This hierarchy of clusters is created through the algorithm that builds the clusters. There are two main types of hierarchical clustering algorithms:

- Agglomerative - Agglomerative clustering techniques start with as many clusters as there are records where each cluster contains just one record. The clusters that are nearest each other are merged together to form the next largest cluster. This merging is continued until a

hierarchy of clusters is built with just a single cluster containing all the records at the top of the hierarchy.

- Divisive - Divisive clustering techniques take the opposite approach from agglomerative techniques. These techniques start with all the records in one cluster and then try to split that cluster into smaller pieces and then in turn to try to split those smaller pieces.

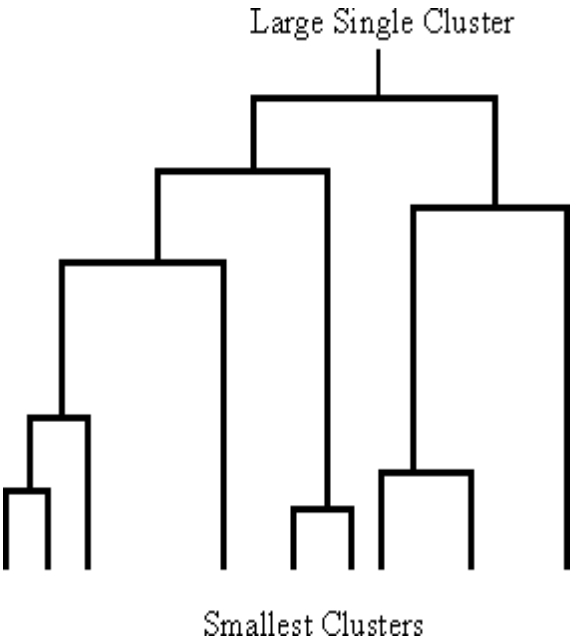


Fig 5.4 hierarchy of clusters

Figure Diagram showing a hierarchy of clusters. Clusters at the lowest level are merged together to form larger clusters at the next level of the hierarchy.

Non-Hierarchical Clustering

There are two main non-hierarchical clustering techniques. Both of them are very fast to compute on the database but have some drawbacks. The first are the single pass methods. They derive their name from the fact that the database must only be passed through once in order to create the clusters (i.e. each record is only read from the database once). The other class of techniques are

called reallocation methods. They get their name from the movement or "reallocation" of records from one cluster to another in order to create better clusters. The reallocation techniques do use multiple passes through the database but are relatively fast in comparison to the hierarchical techniques.

Hierarchical Clustering

Hierarchical clustering has the advantage over non-hierarchical techniques in that the clusters are defined solely by the data (not by the users predetermining the number of clusters) and that the number of clusters can be increased or decreased by simple moving up and down the hierarchy.

The hierarchy is created by starting either at the top (one cluster that includes all records) and subdividing (divisive clustering) or by starting at the bottom with as many clusters as there are records and merging (agglomerative clustering). Usually the merging and subdividing are done two clusters at a time.

The main distinction between the techniques is their ability to favor long, scraggly clusters that are linked together record by record, or to favor the detection of the more classical, compact or spherical cluster that was shown at the beginning of this section. It may seem strange to want to form these long snaking chain like clusters, but in some cases they are the patters that the user would like to have detected in the database. These are the times when the underlying space looks quite different from the spherical clusters and the clusters that should be formed are not based on the distance from the center of the cluster but instead based on the records being "linked" together. Consider the example shown in Figure 1.6 or in Figure 1.7. In these cases there are two clusters that are not very spherical in shape but could be detected by the single link technique.

When looking at the layout of the data in Figure1.6 there appears to be two relatively flat clusters running parallel to each along the income axis. Neither the complete link nor Ward's method would, however, return these two clusters to the user. These techniques rely on creating a "center" for each cluster and picking these centers so that they average distance of each record from this center is minimized. Points that are very distant from these centers would necessarily fall into a different cluster..

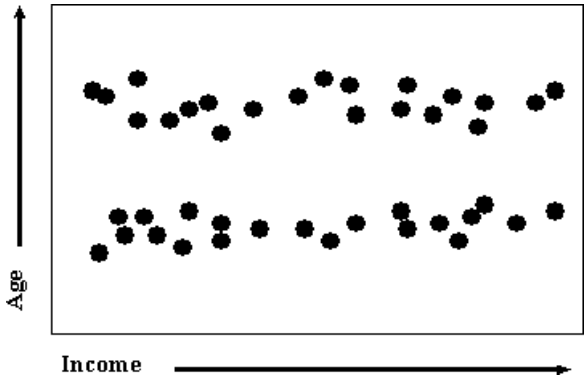


Figure 5.6 an example of elongated clusters

This would not be recovered by the complete link or Ward's methods but would be by the single-link method.

Density-Based Clustering Methods

- Clustering based on density (local cluster criterion), such as density-connected points
- Major features:
  - Discover clusters of arbitrary shape
  - Handle noise
  - One scan
  - Need density parameters as termination condition
- Several interesting studies:
  - DBSCAN: Ester, et al. (KDD'96)
  - OPTICS: Ankerst, et al (SIGMOD'99).
  - DENCLUE: Hinneburg & D. Keim (KDD'98)
  - CLIQUE: Agrawal, et al. (SIGMOD'98)

Density-Based Clustering: Background

- Two parameters*:*
  - *Eps*: Maximum radius of the neighborhood
  - *MinPts*: Minimum number of points in an Eps-neighbour hood of that point
- *NEps(p)*: *{q belongs to D | dist(p,q) <= Eps}*
- Directly density-reachable: A point *p* is directly density-reachable from a point *q* wrt. *Eps*, *MinPts* if
  - *p* belongs to *NEps(q)*

- 2) core point condition:

$|NEps(q)| >= MinPts$

Density-Based Clustering: Background (II)

- Density-reachable:
  - A point *p* is density-reachable from a point *q* wrt. *Eps*, *MinPts* if there is a chain of points *p1*, …, *pn*, *p1 = q*, *pn = p* such that *pi+1* is directly density-reachable from *pi*
- Density-connected
  - A point *p* is density-connected to a point *q* wrt. *Eps*, *MinPts* if there is a point *o*

Such that both, *p* and *q* are density-reachable from *o* wrt. *Eps* and *MinPts*.

DBSCAN: Density Based Spatial Clustering of Applications with Noise

- Relies on a *density-based* notion of cluster: A *cluster* is defined as a maximal set of density-connected points
- Discovers clusters of arbitrary shape in spatial databases with noise

DBSCAN: The Algorithm

- Arbitrary select a point *p*
- Retrieve all points density-reachable from *p* wrt *Eps* and *MinPts*.
- If *p* is a core point, a cluster is formed.
- If *p* is a border point, no points are density-reachable from *p* and DBSCAN visits the next point of the database.
- Continue the process until all of the points have been processed

OPTICS: A Cluster-Ordering Method (1999)

- OPTICS: Ordering Points To Identify the Clustering Structure
- Produces a special order of the database wrt its density-based clustering structure
- This cluster-ordering contains info equiv to the density-based clustering corresponding to a broad range of parameter settings
- Good for both automatic and interactive cluster analysis, including finding intrinsic clustering structure
- Can be represented graphically or using visualization techniques

OPTICS: Some Extension from DBSCAN

- Index-based:
- k = number of dimensions
- N = 20

- p = 75%
- M = N(1-p) = 5
  - Complexity: O(*kN2*)
- Core Distance
- Reachability Distance

Max (core-distance (o), d (o, p))
r(p1, o) = 2.8cm. r(p2,o) = 4cm DENCLUE: using density functions

- Density-based Clustering by Hinneburg & Keim (KDD‘98)
- Major features
  - Solid mathematical foundation
  - Good for data sets with large amounts of noise
  - But needs a large number of parameters

Denclue: Technical Essence

- Uses grid cells but only keeps information about grid cells that do actually contain data points and manages these cells in a tree-based access structure.
- Influence function: describes the impact of a data point within its neighborhood.
- Overall density of the data space can be calculated as the sum of the influence function of all data points.
- Clusters can be determined mathematically by identifying density attractors.
- Density attractors are local maximal of the overall densityfunction.

Grid-Based Methods

Using multi-resolution grid data structure

- Several interesting methods
  -STING (a Statistical Information Grid approach) by Wang, Yang and Muntz (1997)
  -Wave Cluster by Sheikholeslami, Chatterjee, and Zhang (VLDB‘98)
- A multi-resolution clustering approach using wavelet method
  -CLIQUE: Agrawal, et al. (SIGMOD‘98)

STING: A Statistical Information Grid Approach

- Wang, Yang and Muntz (VLDB‘97)
- The spatial area is divided into rectangular cells
- There are several levels of cells corresponding to different levels of resolution

STING: A Statistical Information Grid Approach (2)

- Each cell at a high level is partitioned into a number of smaller cells in the next lower level
- Statistical info of each cell is calculated and stored beforehand and is used to answer queries
- Parameters of higher level cells can be easily calculated from parameters of lower level cell
  - *count*, *mean*, *s*, *min*, *max*
  - type of distribution—normal, *uniform*, etc.
- Use a top-down approach to answer spatial data queries
- Start from a pre-selected layer—typically with a small number of cells
- For each cell in the current level compute the confidence interval

STING: A Statistical Information Grid Approach (3)

- Remove the irrelevant cells from further consideration
- When finish examining the current layer, proceed to the next lower level
- Repeat this process until the bottom layer is reached
- **Advantages:**
  - Query-independent, easy to parallelize, incremental update
  - *O(K)*, where *K* is the number of grid cells at the lowest level
- **Disadvantages:**
  - All the cluster boundaries are either horizontal or vertical, and no diagonal boundary is detected.

Wave Cluster (1998)

- A multi-resolution clustering approach which applies wavelet transform to the feature space
- A wavelet transform is a signal processing technique that decomposes a signal into different frequency sub-band.
- Both grid-based and density-based
- Input parameters:
- # of grid cells for each dimension
- the wavelet, and the # of applications of wavelet transform.
- **How to apply wavelet transform to find clusters**
- Summaries the data by imposing a multidimensional grid structure onto data space
- These multidimensional spatial data objects are represented in a n-dimensional feature space

- Apply wavelet transform on feature space to find the dense regions in the feature space
- Apply wavelet transform multiple times which result in clusters at different scales from fine to coarse

Why is wavelet transformation useful for clustering

- Unsupervised clustering

It uses hat-shape filters to emphasize region where points cluster, but simultaneously to suppress weaker information in their boundary

- Effective removal of outliers
- Multi-resolution
- Cost efficiency
- Major features:
- Complexity O(N)
- Detect arbitrary shaped clusters at differentscales
- Not sensitive to noise, not sensitive to input order
- Only applicable to low dimensional data

Model-Based Clustering Methods:

1.Attempt to optimize the fit between the data and some mathematical model
2.Statistical and AI approach Conceptual clustering
3.A form of clustering in machine learning
4. Produces a classification scheme for a set of unlabeled objects
5. Finds characteristic description for each concept(class) COBWEB (Fisher'87)
6.A popular a simple method of incremental conceptual learning
7.Creates a hierarchical clustering in the form of a classification tree
8. Each node refers to a concept and contains a probabilistic description of that concept Other Model-Based Clustering Methods:

Neural network approaches

Represent each cluster as an exemplar, acting as a "prototype" of the cluster
New objects are distributed to the cluster whose exemplar is the most similar according to some distance measure

Competitive learning

Involves a hierarchical architecture of several units (neurons)
Neurons compete in a "winner-takes-all" fashion for the object currently being presented

CLIQUE (Clustering In QUEst)

- Agrawal, Gehrke, Gunopulos, Raghavan (SIGMOD'98).

- Automatically identifying subspaces of a high dimensional data space that allow better clustering than original space
- CLIQUE can be considered as both density-based and grid-based
  - It partitions each dimension into the same number of equal length interval
  - It partitions an m-dimensional data space into non-overlapping rectangular units.
  - A unit is dense if the fraction of total data points contained in the unit exceeds the input model parameter
- A cluster is a maximal set of connected dense units within a subspace

CLIQUE: The Major Steps

- Partition the data space and find the number of points that lie inside each cell of the partition.
- Identify the subspaces that contain clusters using the Apriori principle
- Identify clusters:
  - Determine dense units in all subspaces of interests
  - Determine connected dense units in all subspaces of interests.
- Generate minimal description for the clusters
  - Determine maximal regions that cover a cluster of connected dense units for each cluster
  - Determination of minimal cover for each cluster

Strength and Weakness of *CLIQUE*

- Strength
  - It *automatically* finds subspaces of the highest dimensionality such that high density clusters exist in those subspaces
  - It is *insensitive* to the order of records in input and does not presume some canonical data distribution
  - It scales *linearly* with the size of input and has good scalability as the number of dimensions in the data increases
- Weakness
  - The accuracy of the clustering result may be degraded at the expense of simplicity of the method

Model-Based Clustering Methods

- Attempt to optimize the fit between the data and some mathematical model
- Statistical and AI approach
  - Conceptual clustering

- A form of clustering in machine learning
- Produces a classification scheme for a set of unlabeled objects
- Finds characteristic description for each concept (class)
- COBWEB (Fisher‘87)
- A popular a simple method of incremental conceptual learning
- Creates a hierarchical clustering in the form of a classification tree
- Each node refers to a concept and contains a probabilistic description of that concept

COBWEB Clustering Method A classification tree

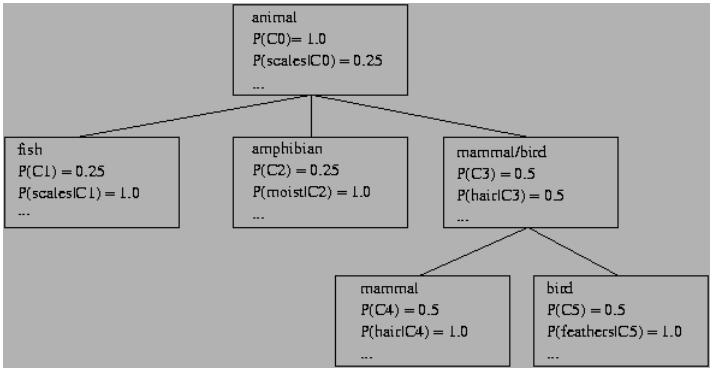


Fig 5.7.COBWEB Clustering Method A classification tree

More on Statistical-Based Clustering

- Limitations of COBWEB
- The assumptionthat the attributes are independent of each other is often too strong because correlation may exist
- Not suitable for clustering large database data – skewed tree and expensive probability distributions
- CLASSIT
- an extension of COBWEB for incremental clustering of continuous data
- suffers similar problems as COBWEB
- AutoClass (Cheeseman and Stutz, 1996)
- Uses Bayesian statistical analysis to estimate the number of clusters
- Popular in industry

Other Model-Based Clustering Methods

- Neural network approaches

- Represent each cluster as an exemplar, acting as a ―prototype‖ of the cluster
- New objects are distributed to the cluster whose exemplar is the most similar according to some distance measure.

- Competitive learning
- Involves a hierarchical architecture of several units (neurons)
- Neurons compete in a ―winner-takes-all‖ fashion for the object currently beingpresented.

Outlier Analysis

What Is Outlier Discovery?

- What are outliers?
- The set of objects are considerably dissimilar from the remainder of the data
- Example: Sports: Michael Jordon, Wayne Gretzky, ...
- Problem
- Find top n outlier points
- Applications:
- Credit card fraud detection
- Telecom fraud detection
- Customer segmentation
- Medical analysis

Outlier Discovery: Statistical Approaches

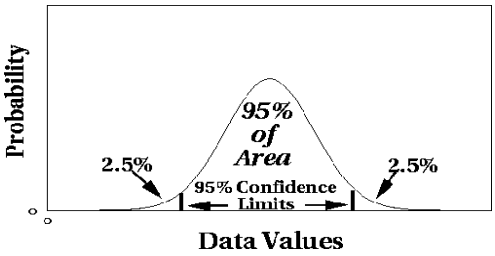


Figure 5.8 outlier Discover

- Assume a model underlying distribution that generates data set (e.g. normal distribution)
- Use discordance tests depending on
- data distribution

- distribution parameter (e.g., mean, variance)
- number of expected outliers
- Drawbacks
- most tests are for single attribute
- In many cases, data distribution may not be known.

Outlier Discovery: Distance-Based Approach

- Introduced to counter the main limitations imposed by statistical methods
- We need multi-dimensional analysis without knowing data distribution.
- Distance-based outlier: A DB(p, D)-outlier is an object O in a dataset T such that at least a fraction p of the objects in T lies at a distance greater than D from O
- Algorithms for mining distance-based outliers
- Index-based algorithm
- Nested-loop algorithm
- Cell-based algorithm

Outlier Discovery: Deviation-Based Approach

- Identifies outliers by examining the main characteristics of objects in a group
- Objects that "deviate" from this description are considered outliers
- sequential exception technique simulates the way in which humans can distinguish unusual objects from among a series of supposedly like objects
- OLAP data cube technique
  - uses data cubes to identify regions of anomalies in large multidimensional data