CBCS SCHEME

USN | | | | | | | | | | |

21AI71

Seventh Semester B.E./B.Tech. Degree Examination, Dec.2024/Jan.2025
Advanced AI and ML

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. Define AI. Explain the foundation of AI in detail. **(10 Marks)**

b. Explain history of AI in detail. **(10 Marks)**

OR

2 a. Briefly explain the properties of task environment. **(10 Marks)**

b. Explain the following with respect to structure of agents:
i) Simple reflex agents
ii) Model based reflex agents
iii) Utility based agents **(10 Marks)**

Module-2

3 a. What is decision theory? Describe the decision theoretic agent that selects rational actions. **(10 Marks)**

b. What is Baye's rule? Explain with a relevant example. **(10 Marks)**

OR

4 a. Explain the following with examples:
i) Kolmogorov's axioms
ii) Inclusion – Exclusion principle
iii) Probability density function
iv) Joint Probability distribution
v) Independence **(10 Marks)**

b. Prove that probabilistic agent can perform better than logical agent by concept of wumpus world. **(10 Marks)**

Module-3

5 a. Define perceptrons. How the perceptrons are represented? Explain perceptron training rule. **(08 Marks)**

b. Derive the gradient descent rule. **(08 Marks)**

c. Write the stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward network containing 2 layers of sigmoid units. **(04 Marks)**

OR

6 a. Write the prototypical genetic algorithm. **(05 Marks)**

b. Explain the different operators with relevant bit strings. **(06 Marks)**

c. Illustrate program tree representation in genetic programming. Explain block stacking problem. **(09 Marks)**

Module-4

7 a. What is association rule mining? Explain support, confidence and lift. **(10 Marks)**

b. What is collaborative filtering? Explain the types. **(10 Marks)**

OR

8 a. What is BOW model? What are the 3 ways to identify the importance of words in BOW model? **(08 Marks)**

b. Explain Naïve – Baye's model for sentiment classification. **(08 Marks)**

c. Brief stemming and lemmatization process. **(04 Marks)**

Module-5

9 a. Define Clustering. What are the different types of clustering? **(06 Marks)**

b. Explain k-medoids clustering with relevant example. **(08 Marks)**

c. Write the k-nearest neighbor algorithm using voronoi diagram. **(06 Marks)**

OR

10 a. Explain distance weighted Nearest neighbor algorithm. **(05 Marks)**

b. Derive and explain locally weighted Linear Regression. **(10 Marks)**

c. Briefly explain radial basis funciton. **(05 Marks)**

* * * * *

Module 1

1. **Define AI. Explain the foundation of AI in detail.**

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole *et al.*, 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Figure 1.1 Some definitions of artificial intelligence, organized into four categories.

Foundation of AI

- Philosophy
- Mathematics
- Economics
- Neuroscienc
- Psychology
- Computer engineering
- Control theory and cybernetics
- Linguistics
-

2. **Explain history of AI in detail**
   - The gestation of artificial intelligence (1943–1955)
   - The birth of artificial intelligence (1956)
   - Early enthusiasm, great expectations (1952–1969)
   - A dose of reality (1966–1973)
   - Knowledge-based systems: The key to power? (1969–1979)
   - AI becomes an industry (1980–present)
   - The return of neural networks (1986–present)
   - AI adopts the scientific method (1987–present)

- The emergence of intelligent agents (1995–present)
- The availability of very large data sets (2001–present)

OR

1. **Brieflr explain the properties of task environment.**

    - Fully observable vs. partially observable
    - Single agent vs. multiagent:
    - Deterministic vs. stochastic
    - Episodic vs. sequential:
    - Static vs. dynamic
    - Discrete vs. continuous:
    -

2. **Explain the following w.r.t. Structure of agents.**
    a. Simplex reflex agenents
    b. Model based reflex agents
    c. Utility based agents

The simplest kind of agent is the **simple reflex agent.** These agents select actions on the basis of the current percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Figure 2.3 is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt. An agent program for this agent is shown in Figure 2.8.

Simple reflex behaviors occur even in more complex environments. Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking. In other words, some processing is done on the visual input to establish the condition we call "The car in front is braking." Then, this triggers some established connection in the agent program to the action "initiate braking." We call such a connection a **condition–action rule**, written as

if car-in-front-is-braking then initiate-braking.

The program in Figure 2.8 is specific to one particular vacuum environment. A more general and flexible approach is first to build a general-purpose interpreter for condition–action rules and then to create rule sets for specific task environments. Figure 2.9 gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background

information used in the process. The agent program, which is also very simple, is shown in Figure 2.10. The INTERPRET-INPUT function generates an abstracted description of the current state from the percept, and the RULE-MATCH function returns the first rule in the set of rules that matches the given state description. Note that the description in terms of "rules" and "matching" is purely conceptual; actual implementations can be as simple as a collection of logic gates implementing a Boolean circuit.

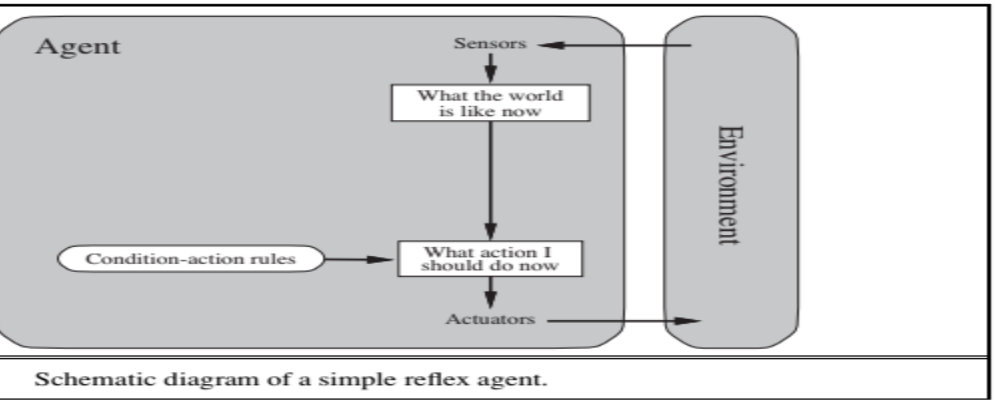


Figure 2.9 Schematic diagram of a simple reflex agent.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

Simple reflex agents have the admirable property of being simple, but they turn out to be of limited intelligence. The agent in Figure 2.10 will work *only if the correct decision can be made on the basis of only the current percept*—that is, only if the environment is fully observable. Even a little bit of unobservability can cause serious trouble. For example, the braking rule given earlier assumes that the condition car-in-front-is-braking can be determined from the current percept—a single frame of video. This works if the car in front has a centrally mounted brake light. Unfortunately, older models have different configurations of taillights, brake lights, and turn-signal lights, and it is not always possible to tell from a single image whether the car is braking. A simple reflex agent driving behind such a car would either brake continuously and unnecessarily, or, worse, never brake at all.

A. Model-based reflex agents

The most effective way to handle partial observability is for the agent to keep track of the part of the world it can't see now. That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. For the braking problem, the internal state is not too extensive—just the previous frame from the camera, allowing the agent to detect when two red lights at the edge of the vehicle go on or off simultaneously. For other driving tasks such as changing lanes, the agent needs to keep track of where the other cars are if it can't see them all at once. And for any driving to be possible at all, the agent needs to keep track of where its keys are.

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program. First, we need some information about how the world evolves independently of the agent—for example, that an overtaking car generally will be closer behind than it was a moment ago. Second, we need some information about how the agent's own actions affect the world—for example, that when the agent turns the steering wheel clockwise, the car turns to the right, or that after driving for five minutes northbound on the freeway, one is usually about five miles north of where one was five minutes ago. This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories—is called a **model** of the world. An agent that uses such a model is called **model-based agent.**

Figure 2.11 gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works. The agent program is shown in Figure 2.12. The interesting part is the function UPDATE-STATE, which is responsible for creating the new internal state description. The details of how models and states are represented vary widely depending on the type of environment and the particular technology used in the agent design.

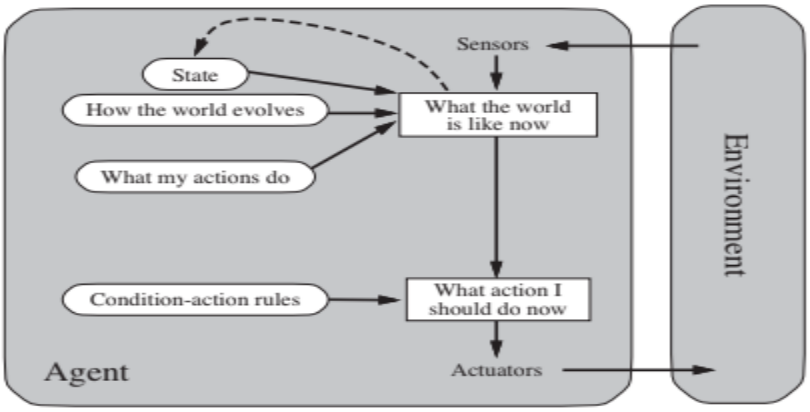


Figure 2.11 A model-based reflex agent.

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              model, a description of how the next state depends on current state and action
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

Regardless of the kind of representation used, it is seldom possible for the agent to determine the current state of a partially observable environment exactly. Instead, the box labeled "what the world is like now" (Figure 2.11) represents the agent's "best guess" (or sometimes best guesses). For example, an automated taxi may not be able to see around the large truck that has stopped in front of it and can only guess about what may be causing the hold-up. Thus, uncertainty about the current state may be unavoidable, but the agent still has to make a decision.

A perhaps less obvious point about the internal "state" maintained by a model-based agent is that it does not have to describe "what the world is like now" in a literal sense. For example, the taxi may be driving back home, and it may have a rule telling it to fill up with gas on the way home unless it has at least half a tank. Although "driving back home" may seem to an aspect of the world state, the fact of the taxi's destination is actually an aspect of the agent's internal state. If you find this puzzling, consider that the taxi could be in exactly the same place at the same time, but intending to reach a different destination.

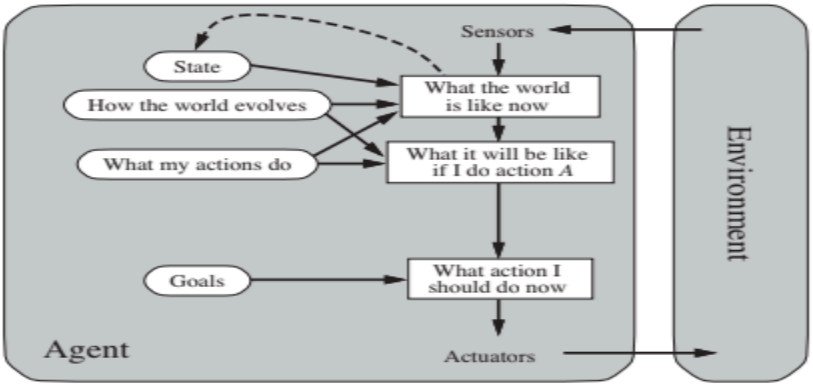


Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between "happy" and "unhappy" states. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. Because "happy" does not sound very scientific, economists and computer scientists use the term utility instead.6 UTILITY We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of UTILITY FUNCTION getting to the taxi's destination. An agent's utility function is essentially an internalization of the performance measure. If the internal utility function and the external performance measure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure. Let us emphasize again that this is not the only way to be rational—we have already seen a rational agent program for the vacuum world (Figure 2.8) that has no idea what its utility function is—but, like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning. Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff. Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals. Partial observability and stochasticity are ubiquitous in the real world, and so, therefore, is decision making under uncertainty. Technically speaking, a rational utility-based agent EXPECTED UTILITY chooses the action that maximizes the expected utility of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome.

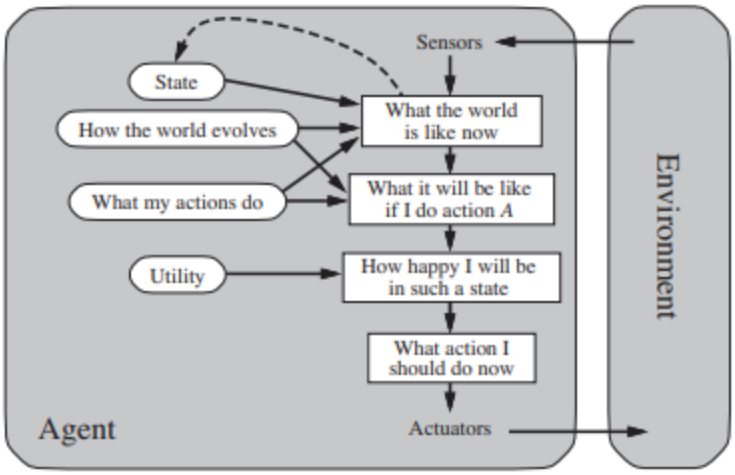


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

Module -2

1. **What is decision theory? Describe the decision theoretic agents that selects rational actions.**

   Decision theory = probability theory + utility theory .

```
function DT-AGENT(percept) returns an action
  persistent: belief_state, probabilistic beliefs about the current state of the world
              action, the agent's action

  update belief_state based on action and percept
  calculate outcome probabilities for actions,
      given action descriptions and current belief_state
  select action with highest expected utility
      given probabilities of outcomes and utility information
  return action
```

Figure 13.1 A decision-theoretic agent that selects rational actions.

2. **What is Bayes rule? Explain with relevant examples.**

$$P(a \wedge b) = P(a \mid b)P(b) \qquad \text{and} \qquad P(a \wedge b) = P(b \mid a)P(a) \ .$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b \mid a) = \frac{P(a \mid b)P(b)}{P(a)} \ . \tag{13.12}$$

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem). This simple equation underlies most modern AI systems for probabilistic inference.

On the surface, Bayes' rule does not seem very useful. It allows us to compute the single term $P(b \mid a)$ in terms of three terms: $P(a \mid b)$, $P(b)$, and $P(a)$. That seems like two steps backwards, but Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth. Often, we perceive as evidence the *effect* of some unknown *cause* and we would like to determine that cause. In that case, Bayes' rule becomes

$$P(\mathit{cause} \mid \mathit{effect}) = \frac{P(\mathit{effect} \mid \mathit{cause})P(\mathit{cause})}{P(\mathit{effect})} \ .$$

The conditional probability $P(\mathit{effect} \mid \mathit{cause})$ quantifies the relationship in the **causal** direction, whereas $P(\mathit{cause} \mid \mathit{effect})$ describes the **diagnostic** direction. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships (that is, the doctor knows $P(\mathit{symptoms} \mid \mathit{disease})$) and want to derive a diagnosis, $P(\mathit{disease} \mid \mathit{symptoms})$. For example, a doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 70% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1%. Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have

$$\begin{aligned} P(s \mid m) &= 0.7 \\ P(m) &= 1/50000 \\ P(s) &= 0.01 \\ P(m \mid s) &= \frac{P(s \mid m)P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} = 0.0014 \ . \end{aligned} \tag{13.14}$$

That is, we expect less than 1 in 700 patients with a stiff neck to have meningitis. Notice that even though a stiff neck is quite strongly indicated by meningitis (with probability 0.7), the probability of meningitis in the patient remains small. This is because the prior probability of stiff necks is much higher than that of meningitis.

OR

1. **Explain the following with example:**
   a. **Kolmogorovs axioms**

$$0 \leq P(\omega) \leq 1 \text{ for every } \omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1 \,. \tag{13.1}$$

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) \,. \tag{13.4}$$

Equations (13.1) and (13.4) are often called Kolmogorov's axioms in honor of the Rus- KOLMOGOROV'S AXIOMS sian mathematician Andrei Kolmogorov, who showed how to build up the rest of probability theory from this simple foundation and how to handle the difficulties caused by continuous variables.2 While Equation (13.2) has a definitional flavor, Equation (13.4) reveals that the axioms really do constrain the degrees of belief an agent can have concerning logically related propositions. This is analogous to the fact that a logical agent cannot simultaneously believe A, B, and ¬(A ∧ B), because there is no possible world in which all three are true. With probabilities, however, statements refer not to the world directly, but to the agent's own state of knowledge

b. Inclusion-Exclusion principle.

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) \,. \tag{13.4}$$

This rule is easily remembered by noting that the cases where a holds, together with the cases where b holds, certainly cover all the cases where a ∨ b holds; but summing the two sets of cases counts their intersection twice, so we need to subtract P(a ∧ b).

c. Probability density function

Probability density functions (sometimes called **pdfs**) differ in meaning from discrete distributions. Saying that the probability density is uniform from $18C$ to $26C$ means that there is a 100% chance that the temperature will fall somewhere in that $8C$-wide region and a 50% chance that it will fall in any $4C$-wide region, and so on. We write the probability density for a continuous random variable X at value x as $P(X = x)$ or just $P(x)$; the intuitive definition of $P(x)$ is the probability that X falls within an arbitrarily small region beginning at x, divided by the width of the region:

$$P(x) = \lim_{dx \to 0} P(x \leq X \leq x + dx)/dx \,.$$

For $NoonTemp$ we have

$$P(NoonTemp = x) = Uniform_{[18C,26C]}(x) = \begin{cases} \frac{1}{8C} \text{ if } 18C \leq x \leq 26C \\ 0 \text{ otherwise} \end{cases} ,$$

where C stands for centigrade (not for a constant). In $P(NoonTemp = 20.18C) = \frac{1}{8C}$, note that $\frac{1}{8C}$ is not a probability, it is a probability density. The probability that $NoonTemp$ is *exactly* $20.18C$ is zero, because $20.18C$ is a region of width 0. Some authors use different symbols for discrete distributions and density functions; we use P in both cases, since confusion seldom arises and the equations are usually identical. Note that probabilities are unitless numbers, whereas density functions are measured with a unit, in this case reciprocal degrees.

d. Joint probability distribution

In addition to distributions on single variables, we need notation for distributions on multiple variables. Commas are used for this. For example, $\mathbf{P}(Weather, Cavity)$ denotes the probabilities of all combinations of the values of $Weather$ and $Cavity$. This is a 4×2 table of probabilities called the **joint probability distribution** of $Weather$ and $Cavity$. We can also mix variables with and without values; $\mathbf{P}(sunny, Cavity)$ would be a two-element vector giving the probabilities of a sunny day with a cavity and a sunny day with no cavity. The **P** notation makes certain expressions much more concise than they might otherwise be.

e. Independence

Let us expand the full joint distribution in Figure 13.3 by adding a fourth variable, $Weather$. The full joint distribution then becomes $\mathbf{P}(Toothache, Catch, Cavity, Weather)$, which has $2 \times 2 \times 2 \times 4 = 32$ entries. It contains four "editions" of the table shown in Figure 13.3, one for each kind of weather. What relationship do these editions have to each other and to the original three-variable table? For example, how are $P(toothache, catch, cavity, cloudy)$ and $P(toothache, catch, cavity)$ related? We can use the product rule:

$$\begin{aligned} &P(toothache, catch, cavity, cloudy) \\ &= P(cloudy \mid toothache, catch, cavity) P(toothache, catch, cavity) \ . \end{aligned}$$

Now, unless one is in the deity business, one should not imagine that one's dental problems influence the weather. And for indoor dentistry, at least, it seems safe to say that the weather does not influence the dental variables. Therefore, the following assertion seems reasonable:

$$P(cloudy \mid toothache, catch, cavity) = P(cloudy) \ . \tag{13.10}$$

From this, we can deduce

$$P(toothache, catch, cavity, cloudy) = P(cloudy) P(toothache, catch, cavity) \ .$$

A similar equation exists for *every entry* in $\mathbf{P}(Toothache, Catch, Cavity, Weather)$. In fact, we can write the general equation

$$\mathbf{P}(Toothache, Catch, Cavity, Weather) = \mathbf{P}(Toothache, Catch, Cavity) \mathbf{P}(Weather) \ .$$

Thus, the 32-element table for four variables can be constructed from one 8-element table and one 4-element table. This decomposition is illustrated schematically in Figure 13.4(a).

The property we used in Equation (13.10) is called **independence** (also **marginal independence** and **absolute independence**). In particular, the weather is independent of one's dental problems. Independence between propositions a and b can be written as

$$P(a \mid b) = P(a) \quad \text{or} \quad P(b \mid a) = P(b) \quad \text{or} \quad P(a \wedge b) = P(a) P(b) \ . \tag{13.11}$$

All these forms are equivalent (Exercise 13.12). Independence between variables X and Y can be written as follows (again, these are all equivalent):

$$\mathbf{P}(X \mid Y) = \mathbf{P}(X) \quad \text{or} \quad \mathbf{P}(Y \mid X) = \mathbf{P}(Y) \quad \text{or} \quad \mathbf{P}(X, Y) = \mathbf{P}(X) \mathbf{P}(Y) \ .$$

Independence assertions are usually based on knowledge of the domain. As the toothache–weather example illustrates, they can dramatically reduce the amount of information necessary to specify the full joint distribution. If the complete set of variables can be divided into independent subsets, then the full joint distribution can be *factored* into separate joint distributions on those subsets. For example, the full joint distribution on the outcome of n independent coin flips, $\mathbf{P}(C_1, \ldots, C_n)$, has 2^n entries, but it can be represented as the product of n single-variable distributions $\mathbf{P}(C_i)$. In a more practical vein, the independence of dentistry and meteorology is a good thing, because otherwise the practice of dentistry might require intimate knowledge of meteorology, and vice versa.

2. Prove the probabilistic agent can perform better than logical agent by concept of wumpus world.

Our aim is to calculate the probability that each of the three squares contains a pit. (For this example we ignore the wumpus and the gold.) The relevant properties of the wumpus world are that (1) a pit causes breezes in all neighboring squares, and (2) each square other than [1,1] contains a pit with probability 0.2. The first step is to identify the set of random variables we need:

- As in the propositional logic case, we want one Boolean variable P_{ij} for each square, which is true iff square $[i, j]$ actually contains a pit.

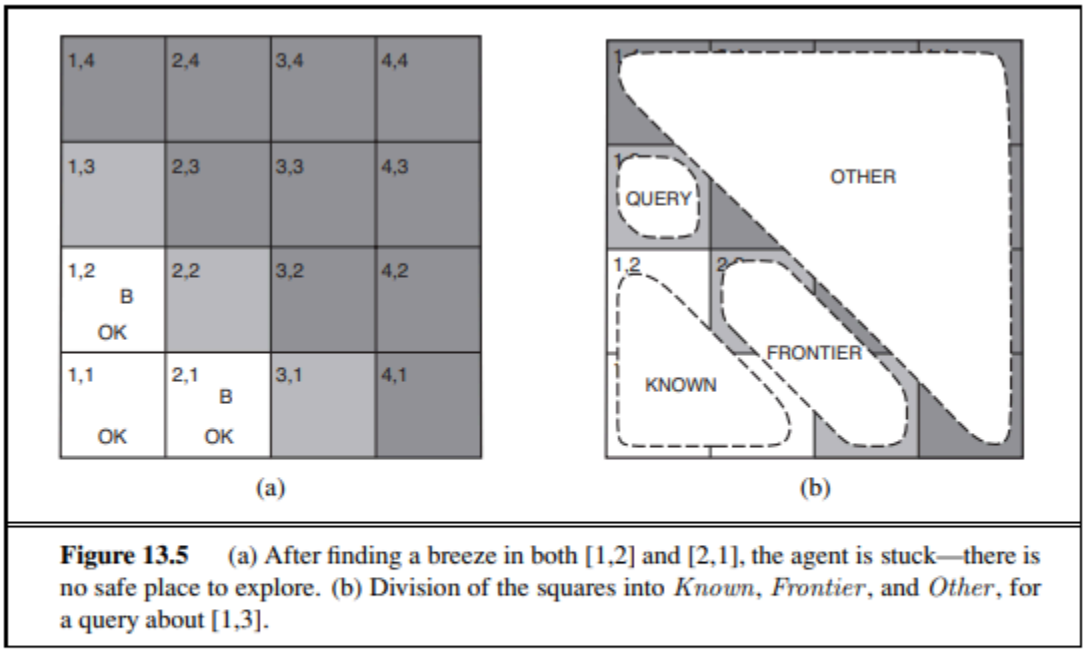


Figure 13.5 (a) After finding a breeze in both [1,2] and [2,1], the agent is stuck—there is no safe place to explore. (b) Division of the squares into *Known*, *Frontier*, and *Other*, for a query about [1,3].

- We also have Boolean variables B_{ij} that are true iff square $[i, j]$ is breezy; we include these variables only for the observed squares—in this case, [1,1], [1,2], and [2,1].

The next step is to specify the full joint distribution, $\mathbf{P}(P_{1,1}, \ldots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1})$. Applying the product rule, we have

The next step is to specify the full joint distribution, $\mathbf{P}(P_{1,1},\ldots,P_{4,4},B_{1,1},B_{1,2},B_{2,1})$. Applying the product rule, we have

$$\mathbf{P}(P_{1,1},\ldots,P_{4,4},B_{1,1},B_{1,2},B_{2,1}) = \\ \mathbf{P}(B_{1,1},B_{1,2},B_{2,1} \mid P_{1,1},\ldots,P_{4,4})\mathbf{P}(P_{1,1},\ldots,P_{4,4}) \,.$$

This decomposition makes it easy to see what the joint probability values should be. The first term is the conditional probability distribution of a breeze configuration, given a pit configuration; its values are 1 if the breezes are adjacent to the pits and 0 otherwise. The second term is the prior probability of a pit configuration. Each square contains a pit with probability 0.2, independently of the other squares; hence,

$$\mathbf{P}(P_{1,1},\ldots,P_{4,4}) = \prod_{i,j=1,1}^{4,4} \mathbf{P}(P_{i,j}) \,. \tag{13.20}$$

For a particular configuration with exactly n pits, $P(P_{1,1},\ldots,P_{4,4}) = 0.2^n \times 0.8^{16-n}$.

In the situation in Figure 13.5(a), the evidence consists of the observed breeze (or its absence) in each square that is visited, combined with the fact that each such square contains no pit. We abbreviate these facts as $b = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1}$ and $known = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$. We are interested in answering queries such as $\mathbf{P}(P_{1,3} \mid known, b)$: how likely is it that [1,3] contains a pit, given the observations so far?

To answer this query, we can follow the standard approach of Equation (13.9), namely, summing over entries from the full joint distribution. Let $Unknown$ be the set of $P_{i,j}$ vari-

ables for squares other than the $Known$ squares and the query square [1,3]. Then, by Equation (13.9), we have

$$\mathbf{P}(P_{1,3} \mid known, b) = \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, unknown, known, b) \ .$$

The full joint probabilities have already been specified, so we are done—that is, unless we care about computation. There are 12 unknown squares; hence the summation contains $2^{12} = 4096$ terms. In general, the summation grows exponentially with the number of squares.

Surely, one might ask, aren't the other squares irrelevant? How could [4,4] affect whether [1,3] has a pit? Indeed, this intuition is correct. Let $Frontier$ be the pit variables (other than the query variable) that are adjacent to visited squares, in this case just [2,2] and [3,1]. Also, let $Other$ be the pit variables for the other unknown squares; in this case, there are 10 other squares, as shown in Figure 13.5(b). The key insight is that the observed breezes are *conditionally independent* of the other variables, given the known, frontier, and query variables. To use the insight, we manipulate the query formula into a form in which the breezes are conditioned on all the other variables, and then we apply conditional independence:

$$\begin{aligned}
&\mathbf{P}(P_{1,3} \mid known, b) \\
&= \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, known, b, unknown) \qquad \text{(by Equation (13.9))} \\
&= \alpha \sum_{unknown} \mathbf{P}(b \mid P_{1,3}, known, unknown)\mathbf{P}(P_{1,3}, known, unknown) \\
&\qquad\qquad \text{(by the product rule)} \\
&= \alpha \sum_{frontier} \sum_{other} \mathbf{P}(b \mid known, P_{1,3}, frontier, other)\mathbf{P}(P_{1,3}, known, frontier, other) \\
&= \alpha \sum_{frontier} \sum_{other} \mathbf{P}(b \mid known, P_{1,3}, frontier)\mathbf{P}(P_{1,3}, known, frontier, other) \ ,
\end{aligned}$$

where the final step uses conditional independence: b is independent of $other$ given $known$, $P_{1,3}$, and $frontier$. Now, the first term in this expression does not depend on the $Other$ variables, so we can move the summation inward:

$$\begin{aligned}
&\mathbf{P}(P_{1,3} \mid known, b) \\
&= \alpha \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier) \sum_{other} \mathbf{P}(P_{1,3}, known, frontier, other) \ .
\end{aligned}$$

By independence, as in Equation (13.20), the prior term can be factored, and then the terms can be reordered:

$$\begin{aligned}
&\mathbf{P}(P_{1,3} \mid known, b) \\
&= \alpha \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier) \sum_{other} \mathbf{P}(P_{1,3})P(known)P(frontier)P(other) \\
&= \alpha \, P(known)\mathbf{P}(P_{1,3}) \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier)P(frontier) \sum_{other} P(other) \\
&= \alpha' \, \mathbf{P}(P_{1,3}) \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier)P(frontier) \ ,
\end{aligned}$$

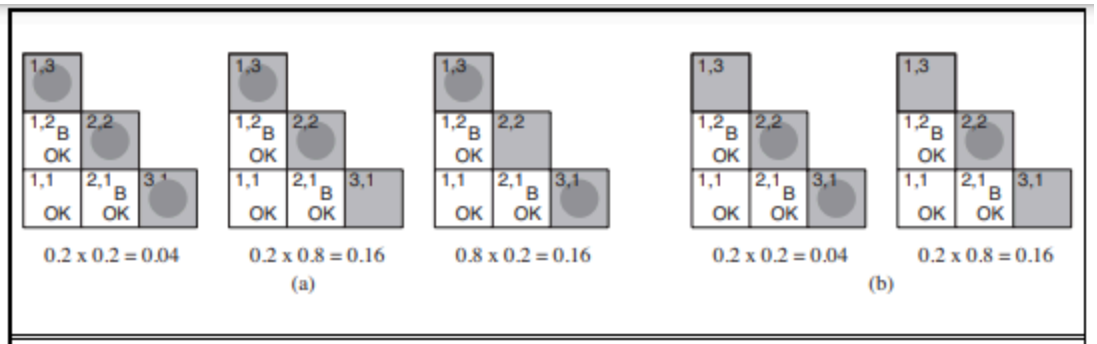


Figure 13.6 Consistent models for the frontier variables $P_{2,2}$ and $P_{3,1}$, showing $P(frontier)$ for each model: (a) three models with $P_{1,3} = true$ showing two or three pits, and (b) two models with $P_{1,3} = false$ showing one or two pits.

where the last step folds $P(known)$ into the normalizing constant and uses the fact that $\sum_{other} P(other)$ equals 1.

Now, there are just four terms in the summation over the frontier variables $P_{2,2}$ and $P_{3,1}$. The use of independence and conditional independence has completely eliminated the other squares from consideration.

Notice that the expression $\mathbf{P}(b \mid known, P_{1,3}, frontier)$ is 1 when the frontier is consistent with the breeze observations, and 0 otherwise. Thus, for each value of $P_{1,3}$, we sum over the *logical models* for the frontier variables that are consistent with the known facts. (Compare with the enumeration over models in Figure 7.5 on page 241.) The models and their associated prior probabilities—$P(frontier)$—are shown in Figure 13.6. We have

$$\mathbf{P}(P_{1,3} \mid known, b) = \alpha' \langle 0.2(0.04 + 0.16 + 0.16),\ 0.8(0.04 + 0.16) \rangle \approx \langle 0.31, 0.69 \rangle .$$

$$\mathbf{P}(P_{1,3} \mid known, b) = \alpha' \langle 0.2(0.04 + 0.16 + 0.16),\ 0.8(0.04 + 0.16) \rangle \approx \langle 0.31, 0.69 \rangle .$$

That is, [1,3] (and [3,1] by symmetry) contains a pit with roughly 31% probability. A similar calculation, which the reader might wish to perform, shows that [2,2] contains a pit with roughly 86% probability. The wumpus agent should definitely avoid [2,2]! Note that our logical agent from Chapter 7 did not know that [2,2] was worse than the other squares. Logic can tell us that it is unknown whether there is a pit in [2, 2], but we need probability to tell us how likely it is.

What this section has shown is that even seemingly complicated problems can be formulated precisely in probability theory and solved with simple algorithms. To get *efficient* solutions, independence and conditional independence relationships can be used to simplify the summations required. These relationships often correspond to our natural understanding of how the problem should be decomposed. In the next chapter, we develop formal representations for such relationships as well as algorithms that operate on those representations to perform probabilistic inference efficiently.

OR

1. **Define perceptrons. How the perceptrons are represented? Explain perceptron training rule.**

One type of ANN system is based on a unit called a *perceptron*, illustrated in Figure 4.2. 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 x_1 through x_n, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

We can view the perceptron as representing a hyperplane 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 hyperplane and outputs a -1 for instances lying on the other side, as illustrated in Figure 4.3. The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separable* sets of examples.

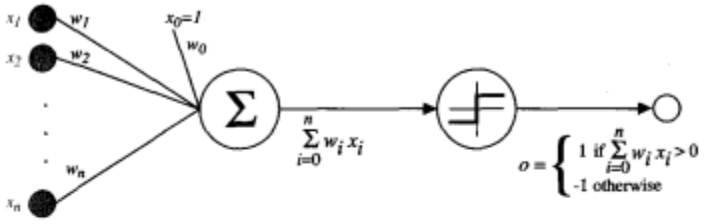


FIGURE 4.2
A perceptron.

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 $w_0 = -.8$, and $w_1 = w_2 = .5$. This perceptron can be made to represent the OR function instead by altering the threshold to $w_0 = -.3$. In fact, AND and OR can be viewed as special cases of m-of-n functions: that is, functions where at least m of the n inputs to the perceptron must be true. The OR function corresponds to $m = 1$ 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 w_0 accordingly.

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 ± 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 guaranteed 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, modifying 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 w_i associated with input x_i according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Here t is the target output for the current training example, o is the output generated by the perceptron, and η is a positive constant called the *learning rate*. The role of the learning rate is to moderate the degree 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 Δw_i 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 $\vec{w} \cdot \vec{x}$. For example, if $x_i > 0$, then increasing w_i will bring the perceptron closer to correctly classifying this example. Notice the training rule will increase w_i in this case, because $(t - o)$, η, and x_i are all positive. For example, if $x_i = .8$, $\eta = 0.1, t = 1$, and $o = -1$, then the weight update will be $\Delta w_i = \eta(t - o)x_i = 0.1(1 - (-1))0.8 = 0.16$. On the other hand, if $t = -1$ and $o = 1$, then weights associated with positive x_i 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 η is used (see Minsky and Papert 1969). If the data are not linearly separable, convergence is not assured.

2. Derive 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 $\vec{w}$. This vector derivative is called the *gradient* of E with respect to $\vec{w}$, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right] \tag{4.3}$$

Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i. *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 Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_0, w_1 plane.

Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

where

$$\Delta \vec{w} = -\eta \nabla E(\vec{w}) \tag{4.4}$$

Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search. 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

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{4.5}$$

which makes it clear that steepest descent is achieved by altering each component w_i of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

To construct a practical algorithm for iteratively updating weights according to Equation (4.5), we need an efficient way of calculating the gradient at each step. Fortunately, this is not difficult. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the

gradient can be obtained by differentiating E from Equation (4.2), as

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x_d})$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id}) \tag{4.6}$$

where x_{id} denotes the single input component x_i for training example d. We now have an equation that gives $\frac{\partial E}{\partial w_i}$ in terms of the linear unit inputs x_{id}, outputs O_d, and target values t_d associated with the training examples. Substituting Equation (4.6) into Equation (4.5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\ x_{id} \tag{4.7}$$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\ x_{id} \tag{4.7}$$

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, then compute Δw_i for each weight according to Equation (4.7). Update each weight w_i by adding Δw_i, then repeat this process. This algorithm is given in Table 4.1. 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 η is used. If η 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 η as the number of gradient descent steps grows.

3. **Write the stochastic gradient descent version of the back propagation algorithm for FFN continuing 2 layers of sigmoid units.**

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $(\vec{x}, \vec{t})$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $(\vec{x}, \vec{t})$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

4.

OR

1. What is prototypical genetic algorithm

GA($Fitness, Fitness\_threshold, p, r, m$)

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

$Fitness\_threshold$: A threshold specifying the termination criterion.

p: The number of hypotheses to be included in the population.

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

m: The mutation rate.

- *Initialize population:* $P \leftarrow$ Generate p hypotheses at random
- *Evaluate:* For each h in P, compute $Fitness(h)$
- While $[\max_h Fitness(h)] < Fitness\_threshold$ do

Create a new generation, P_S:

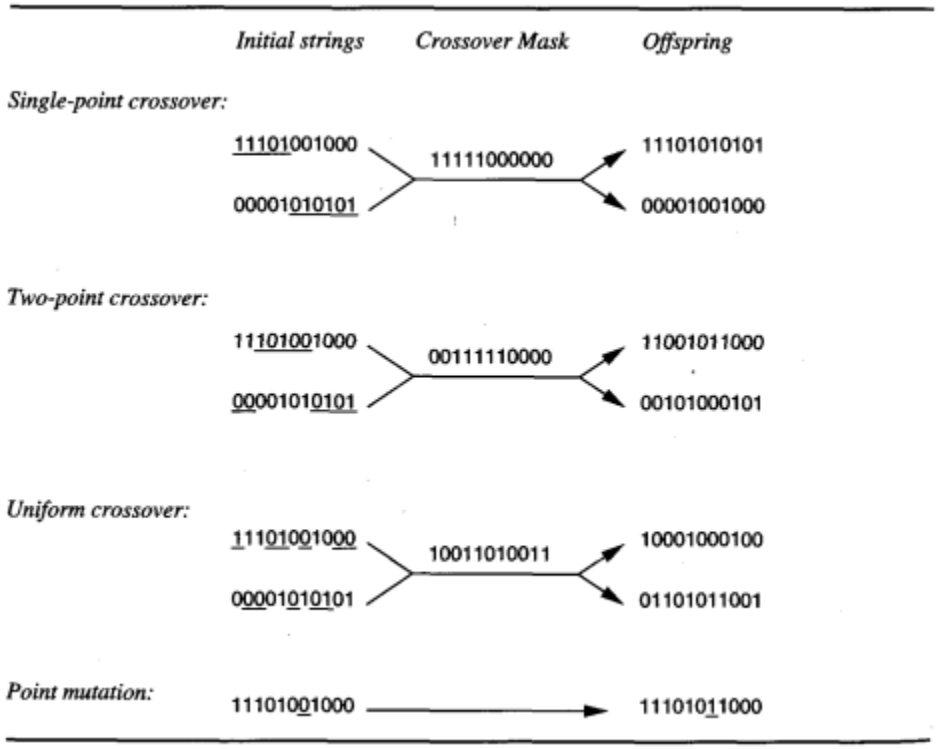
1. *Select:* Probabilistically select $(1 - r)p$ members of P to add to P_S. The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$$

2. *Crossover:* Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P, according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s.
3. *Mutate:* Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.
4. *Update:* $P \leftarrow P_s$.
5. *Evaluate:* for each h in P, compute $Fitness(h)$

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

2. **Explain the different operators with relevant bit strings.**



3. **Illustrate program tree representation in genetic programming. Explain block stacking problem**.

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, Figure 9.1 illustrates this tree representation for the function $\sin(x) + \sqrt{x^2 + y}$. To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., sin, cos, $\sqrt{}$, $+$, $-$, exponentials), 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 subtree of one parent

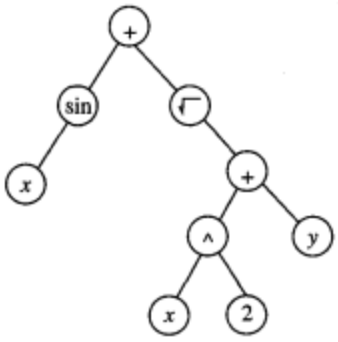
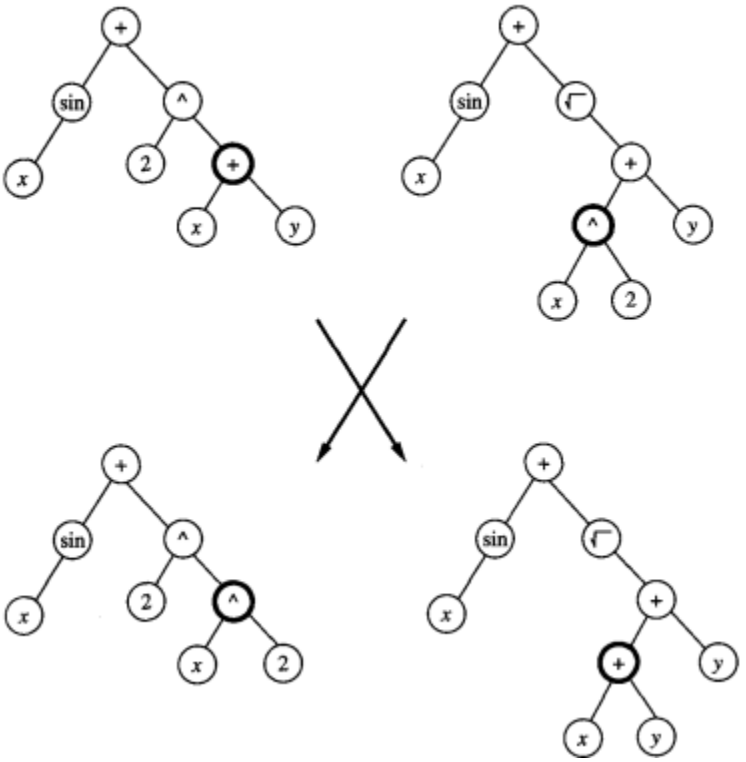
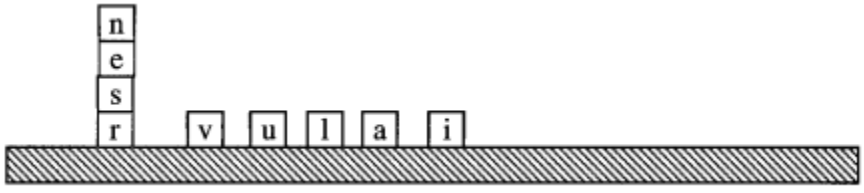


FIGURE 9.1
Program tree representation in genetic programming. Arbitrary programs are represented by their parse trees.



One illustrative example presented by Koza (1992) involves learning an algorithm for stacking the blocks shown in Figure 9.3. The task is to develop a general algorithm for stacking the blocks into a single stack that spells the word "universal,"



independent of the initial configuration of blocks in the world. The actions available for manipulating blocks allow moving only a single block at a time. In particular, the top block on the stack can be moved to the table surface, or a block on the table surface can be moved to the top of the stack.

As in most GP applications, the choice of problem representation has a significant impact on the ease of solving the problem. In Koza's formulation, the primitive functions used to compose programs for this task include the following three terminal arguments:

- CS (current stack), which refers to the name of the top block on the stack, or F if there is no current stack.
- TB (top correct block), which refers to the name of the topmost block on the stack, such that it and those blocks beneath it are in the correct order.
- NN (next necessary), which refers to the name of the next block needed above TB in the stack, in order to spell the word "universal," or F if no more blocks are needed.

As can be seen, this particular choice of terminal arguments provides a natural representation for describing programs for manipulating blocks for this task. Imagine, in contrast, the relative difficulty of the task if we were to instead define the terminal arguments to be the x and y coordinates of each block.

In addition to these terminal arguments, the program language in this application included the following primitive functions:

- (MS x) (move to stack), if block x is on the table, this operator moves x to the top of the stack and returns the value T. Otherwise, it does nothing and returns the value F.
- (MT x) (move to table), if block x is somewhere in the stack, this moves the block at the top of the stack to the table and returns the value T. Otherwise, it returns the value F.
- (EQ x y) (equal), which returns T if x equals y, and returns F otherwise.
- (NOT x), which returns T if $x = F$, and returns F if $x = T$.

- (DU x y) (do until), which executes the expression x repeatedly until expression y returns the value T.

To allow the system to evaluate the fitness of any given program, Koza provided a set of 166 training example problems representing a broad variety of initial block configurations, including problems of differing degrees of difficulty. The fitness of any given program was taken to be the number of these examples solved by the algorithm. The population was initialized to a set of 300 random programs. After 10 generations, the system discovered the following program, which solves all 166 problems.

(EQ (DU (MT CS)(NOT CS)) (DU (MS NN)(NOT NN)))

Notice this program contains a sequence of two DU, or "Do Until" statements. The first repeatedly moves the current top of the stack onto the table, until the stack becomes empty. The second "Do Until" statement then repeatedly moves the next necessary block from the table onto the stack. The role played by the top level EQ expression here is to provide a syntactically legal way to sequence these two "Do Until" loops.

Somewhat surprisingly, after only a few generations, this GP was able to discover a program that solves all 166 training problems. Of course the ability of the system to accomplish this depends strongly on the primitive arguments and functions provided, and on the set of training example cases used to evaluate fitness.

Module 4

1. What is association rule mining? Explain support, confidence and lift.

Association rule finds combinations of items that frequently occur together in orders or baskets (in a retail context). The items that frequently occur together are called *itemsets*. Itemsets help to discover relationships between items that people buy together and use that as a basis for creating strategies like combining products as combo offer or place products next to each other in retail shelves to attract customer attention. An application of association rule mining is in Market Basket Analysis (MBA). MBA is a technique used mostly by retailers to find associations between items purchased by customers.

9.2.1.1 Support

Support indicates the frequencies of items appearing together in baskets with respect to all possible baskets being considered (or in a sample). For example, the support for (beer, diaper) will be 2/4 (based on the data shown in Figure 9.1), that is, 50% as it appears together in 2 baskets out of 4 baskets.

Assume that X and Y are items being considered. Let

1. N be the total number of baskets.
2. N_{XY} represent the number of baskets in which X and Y appear together.
3. N_X represent the number of baskets in which X appears.
4. N_Y represent the number of baskets in which Y appears.

Then the support between X and Y, Support(X, Y), is given by

$$\text{Support}(X, Y) = \frac{N_{XY}}{N} \tag{9.1}$$

To filter out stronger associations, we can set a minimum support (for example, minimum support of 0.01). This means the itemset must be present in at least 1% of baskets. Apriori algorithm uses minimum support criteria to reduce the number of possible itemset combinations, which in turn reduces computational requirements.

If minimum support is set at 0.01, an association between X and Y will be considered *if and only if* both X and Y have minimum support of 0.01. Hence, *apriori* algorithm computes support for each item independently and eliminates items with support less than minimum support. The support of each individual item can be calculated using Eq. (9.1).

9.2.1.2 Confidence

Confidence measures the proportion of the transactions that contain X, which also contain Y. X is called antecedent and Y is called consequent. Confidence can be calculated using the following formula:

$$\text{Confidence}(X \rightarrow Y) = P(Y \mid X) = \frac{N_{XY}}{N_X} \tag{9.2}$$

where $P(Y|X)$ is the conditional probability of Y given X.

9.2.1.3 Lift

Lift is calculated using the following formula:

$$\text{Lift} = \frac{\text{Support}(X, Y)}{\text{Support}(X) \times \text{Support}(Y)} = \frac{N_{XY}}{N_X N_Y} \tag{9.3}$$

Lift can be interpreted as the degree of association between two items. Lift value 1 indicates that the items are independent (no association), lift value of less than 1 implies that the products are substitution (purchase one product will decrease the probability of purchase of the other product) and lift value of greater than 1 indicates purchase of Product X will increase the probability of purchase of Product Y. Lift value of greater than 1 is a necessary condition of generating association rules.

2. What is collaborative filtering explain its types.

Collaborative filtering comes in two variations:

1. **User-Based Similarity:** Finds K similar users based on common items they have bought.
2. **Item-Based Similarity:** Finds K similar items based on common users who have bought those items.

Collaborative filtering is based on the notion of similarity (or distance). For example, if two users **A** and **B** have purchased the same products and have rated them similarly on a common rating scale, then **A** and **B** can be considered similar in their buying and preference behavior. Hence, if **A** buys a new product and rates high, then that product can be recommended to **B**. Alternatively, the products that **A** has already bought and rated high can be recommended to **B**, if not already bought by **B**.

OR

1. What is BOW model? What are the three ways to identify the importance of words in BOW model?

Unlike structured data, features (independent variables) are not explicitly available in text data. Thus, we need to use a process to extract features from the text data. One way is to consider each word as a feature and find a measure to capture whether a word exists or does not exist in a sentence. This is called the bag-of-words (BoW) model

There are three ways to identify the importance of words in a BoW model:

1. Count Vector Model
2. Term Frequency Vector Model
3. Term Frequency-Inverse Document Frequency (TF-IDF) Mode

2. Explain Naive-Bayes model for sentiment classification.

Assume that we would like to predict whether the probability of a document is positive (or negative) given that the document contains a word *awesome*. This can be computed if the probability of the word *awesome* appearing in a document given that it is a positive (or negative) sentiment multiplied by the probability of the document being positive (or negative).

$$P(doc = +ve \mid word = awesome) \propto P(word = awesome \mid doc = +ve) * P(doc = +ve)$$

The posterior probability of the sentiment is computed from the **prior** probabilities of all the words it contains. The assumption is that the occurrences of the words in a document are considered independent and they do not influence each other. So, if the document contains N words and words are represented as W_1, W_2, …., W_N, then

$$P(doc = +ve \mid word = W_1, W_2, \ldots W_N) \propto \prod_{i=1}^{N} P_i(word = W_i \mid doc = +ve) * P(doc = +ve)$$

sklearn.naive_bayes provides a class *BernoulliNB* which is a Naïve–Bayes classifier for multivariate Bernoulli models. BernoulliNB is designed for Binary/Boolean features (feature is either present or absent), which is the case here.

The steps involved in using Naïve–Bayes Model for sentiment classification are as follows:

1. Split dataset into train and validation sets.
2. Build the Naïve–Bayes model.
3. Find model accuracy.

```
from sklearn.naive_bayes import BernoulliNB

nb_clf = BernoulliNB()
nb_clf.fit(train_X.toarray(), train_y)
```

3. Brief stemming and lemmatization process.

Many words appear in multiple forms. For example, love and loved. The vectorizer treats the two words as two separate words and hence creates two separate features. But, if a word has similar meaning in all its form, we can use only the root word as a feature. Stemming and Lemmatization are two popular techniques that are used to convert the words into root words. 1. Stemming: This removes the differences between inflected forms of a word to reduce each word to its root form. This is done by mostly chopping off the end of words (suffix). For instance, love or loved will be reduced to the root word love. The root form of a word may not even be a real word. For example, awesome and awesomeness will be stemmed to awesom. One problem with stemming is that chopping of words may result in words that are not part of vocabulary (e.g., awesom). PorterStemmer and LancasterStemmer are two popular algorithms for stemming, which have rules on how to chop off a word.

2. Lemmatization: This takes the morphological analysis of the words into consideration. It uses a language dictionary (i.e., English dictionary) to convert the words to the root word. For example, stemming would fail to differentiate between man and men, while lemmatization can bring these words to its original form man.

Natural Language Toolkit (NLTK) is a very popular library in Python that has an extensive set of features for natural language processing. NLTK supports PorterStemmer, EnglishStemmer, and LancasterStemmer for stemming, while WordNetLemmatizer for lemmatization. These features can be used in CountVectorizer, while creating count vectors. We need to create a utility method, which takes documents, tokenizes it to create words, stems the words and remove the stop words before returning the final set of words for creating vectors

```
from nltk.stem.snowball import PorterStemmer

stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()

#Custom function for stemming and stop word removal

def stemmed_words(doc):
    ### Stemming of words
    stemmed_words = [stemmer.stem(w) for w in analyzer(doc)]
    ### Remove the words in stop words list
    non_stop_words = [word for word in stemmed_words if not in my_
                      stop_words]
    return non_stop_words
```

CountVectorizer takes a custom analyzer for stemming and stop word removal, before creating count vectors. So, the custom function *stemmed_words()* is passed as an analyzer.

```
count_vectorizer = CountVectorizer(analyzer=stemmed_words,
                                   max_features = 1000)
feature_vector = count_vectorizer.fit(train_ds.text)
train_ds_features = count_vectorizer.transform(train_ds.text)
features = feature_vector.get_feature_names()
features_counts = np.sum(train_ds_features.toarray(), axis = 0)
feature_counts = pd.DataFrame(dict(features = features,
                                   counts = features_counts))
feature_counts.sort_values("counts", ascending = False)[0:15]
```

Module 5

1. **Define Clusterin. What are different types of clustering?**

   Clustering is the process of grouping together data objects into multiple sets or clusters, so that objects within a cluster have high similarity as compared to objects outside of it
   Clustering algorithms can be classified into two main subgroups:
   1. Hard clustering: Each data point either belongs to a cluster completely or not.
   2. Soft clustering: Instead of putting each data point into a separate cluster, a probability or likelihood of that data point to be in those clusters is assigned.

   Clustering algorithms can also be classified as follows:

1. Partitioning method.
2. Hierarchical method.
3. Density-based method
4. Grid-based method

2. Explain k-medoids clustering with relevant example.

The k-medoids algorithm is a clustering algorithm very similar to the k-means algorithm. Both k-means and k-medoids algorithms are partitional and try to minimize the distance between points and cluster center. In contrast to the k-means algorithm, k-medoids chooses data points as centers and uses Manhattan distance to define the distance between cluster centers and data points. This technique clusters the dataset of n objects into k clusters, where the number of clusters k is known in prior. It is more robust to noise and outliers as compared to k-means because it minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances. A medoid is defined as an object of a cluster whose average dissimilarity to all the objects in the cluster is minimal.

The Manhattan distance between two vectors in an n-dimensional real vector space is given by Eq. (13.2). It is used in computing the distance between a data point and its cluster center.

$$d_1(p,q) = \|p-q\|_1 = \sum_{i=1}^{n} |p_i - q_i| \tag{13.2}$$

The most common algorithm in k-medoid clustering is Partitioning Around Medoids (PAM) algorithm. PAM uses a greedy search which is faster than the exhaustive search and may not find the optimum solution. It works as follows:

1. Initialize: select k of the n data points as the medoids.
2. Associate each data point to the closest medoid.
3. While the cost of the configuration decreases: For each medoid m and for each non-medoid data point o:
   - Swap m and o, recompute the cost (sum of distances of points to their medoid).
   - If the total cost of the configuration increased in the previous step, undo the swap.

Cluster the following dataset of 6 objects into two clusters, that is, $k = 2$.

X1	2	6
X2	3	4
X3	3	8
X4	4	2
X5	6	2
X6	6	4

Solution:

Step 1: Two observations c1 = X2 = (3, 4) and c2 = X6 = (6, 4) are randomly selected as medoids (cluster centers).

Step 2: Manhattan distances are calculated to each center to associate each data object to its nearest medoid.

Data Object		*Distance To*	
Sample	***Point***	***c1 = (3, 4)***	***c2 = (6, 4)***
X1	(2, 6)	3	6
X2	(3, 4)	0	3
X3	(3, 8)	4	7
X4	(4, 2)	3	4
X5	(6, 2)	5	2
X6	(6, 4)	3	0
Cost		10	2

Step 3: We select one of the non-medoids O′. Let us assume O′ = (6, 2). So now the medoids are c1(3, 4) and O′(6, 2). If c1 and O′ are the new medoids. We calculate the total cost involved.

Data Object		*Distance To*	
Sample	***Point***	***c1 = (3, 4)***	***c2 = (6, 2)***
X1	(2, 6)	3	8
X2	(3, 4)	0	5
X3	(3, 8)	4	9
X4	(4, 2)	3	2
X5	(6, 2)	5	0
X6	(6, 4)	3	2
Cost		7	4

So cost of swapping medoid from c2 to O′ is 11. Since the cost is less, this is considered as a better cluster assignment. Here swapping is done as the cost is less.

Step 4: We select another non-medoid O′. Let us assume O′ = (4, 2). So now the medoids are c1(3, 4) and O′(4, 2). If c1 and O′ are new medoids, we calculate the total cost involved.

Data Object		*Distance To*	
Sample	*Point*	*c1 = (3, 4)*	*c2 = (4, 2)*
X1	(2, 6)	3	6
X2	(3, 4)	0	3
X3	(3, 8)	4	7
X4	(4, 2)	3	0
X5	(6, 2)	5	2
X6	(6, 4)	3	4
Cost		7	8

So cost of swapping medoid from c2 to O′ is 15. Since the cost is more, this cluster assignment is not considered and the swapping is not done.

Thus, we try other non-medoids points to get minimum cost. The assignment with minimum cost is considered the best. For some applications, *k*-medoids show better results than *k*-means. The most time-consuming part of the *k*-medoids algorithm is the calculation of the distances between objects. The distances matrix can be computed in advance to speed-up the process.

3. Write the k-nearest neighbour using voronoi diagram

Training algorithm:

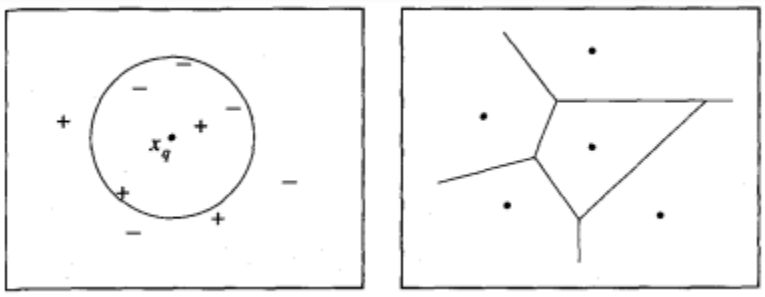
- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.



OR

1. Explain distance weighted nearest neighbour algorithm.

One obvious refinement to the k-NEAREST NEIGHBOR algorithm is to weight the contribution of each of the k neighbors according to their distance to the query point x_q, giving greater weight to closer neighbors. For example, in the algorithm of Table 8.1, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from x_q.

This can be accomplished by replacing the final line of the algorithm by

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i)) \tag{8.2}$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2} \tag{8.3}$$

To accommodate the case where the query point x_q exactly matches one of the training instances x_i and the denominator $d(x_q, x_i)^2$ is therefore zero, we assign $\hat{f}(x_q)$ to be $f(x_i)$ in this case. If there are several such training examples, we assign the majority classification among them.

We can distance-weight the instances for real-valued target functions in a similar fashion, replacing the final line of the algorithm in this case by

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \tag{8.4}$$

where w_i is as defined in Equation (8.3). Note the denominator in Equation (8.4) is a constant that normalizes the contributions of the various weights (e.g., it assures that if $f(x_i) = c$ for all training examples, then $\hat{f}(x_q) \leftarrow c$ as well).

Note all of the above variants of the k-NEAREST NEIGHBOR algorithm consider only the k nearest neighbors to classify the query point. Once we add distance weighting, there is really no harm in allowing all training examples to have an influence on the classification of the x_q, because very distant examples will have very little effect on $\hat{f}(x_q)$. The only disadvantage of considering all examples is that our classifier will run more slowly. If all training examples are considered when classifying a new query instance, we call the algorithm a *global* method. If only the nearest training examples are considered, we call it a *local* method. When the rule in Equation (8.4) is applied as a global method, using all training examples, it is known as Shepard's method (Shepard 1968).

2. Derive and explain local weighted local regression.

Let us consider the case of locally weighted regression in which the target function f is approximated near x_q using a linear function of the form

$$\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$$

As before, $a_i(x)$ denotes the value of the ith attribute of the instance x.

Recall that in Chapter 4 we discussed methods such as gradient descent to find the coefficients $w_0 \ldots w_n$ to minimize the error in fitting such linear functions to a given set of training examples. In that chapter we were interested in a global approximation to the target function. Therefore, we derived methods to choose weights that minimize the squared error summed over the set D of training examples

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \tag{8.5}$$

which led us to the gradient descent training rule

$$\Delta w_j = \eta \sum_{x \in D} (f(x) - \hat{f}(x)) a_j(x) \tag{8.6}$$

where η is a constant learning rate, and where the training rule has been re-expressed from the notation of Chapter 4 to fit our current notation (i.e., $t \rightarrow f(x)$, $o \rightarrow \hat{f}(x)$, and $x_j \rightarrow a_j(x)$).

How shall we modify this procedure to derive a local approximation rather than a global one? The simple way is to redefine the error criterion E to emphasize fitting the local training examples. Three possible criteria are given below. Note we write the error $E(x_q)$ to emphasize the fact that now the error is being defined as a function of the query point x_q.

1. Minimize the squared error over just the k nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2$$

2. Minimize the squared error over the entire set D of training examples, while weighting the error of each training example by some decreasing function K of its distance from x_q:

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

Criterion two is perhaps the most esthetically pleasing because it allows every training example to have an impact on the classification of x_q. However,

this approach requires computation that grows linearly with the number of training examples. Criterion three is a good approximation to criterion two and has the advantage that computational cost is independent of the total number of training examples; its cost depends only on the number k of neighbors considered.

If we choose criterion three above and rederive the gradient descent rule using the same style of argument as in Chapter 4, we obtain the following training rule (see Exercise 8.1):

$$\Delta w_j = \eta \sum_{x \in k \text{ nearest nbrs of } x_q} K(d(x_q, x))\ (f(x) - \hat{f}(x))\ a_j(x) \qquad (8.7)$$

Notice the only differences between this new rule and the rule given by Equation (8.6) are that the contribution of instance x to the weight update is now multiplied by the distance penalty $K(d(x_q, x))$, and that the error is summed over only the k nearest training examples. In fact, if we are fitting a linear function to a fixed set of training examples, then methods much more efficient than gradient descent are available to directly solve for the desired coefficients $w_0 \ldots w_n$. Atkeson et al. (1997a) and Bishop (1995) survey several such methods.

3. Briefly explain radial basis function.

One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions (Powell 1987; Broomhead and Lowe 1988; Moody and Darken 1989). In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \qquad (8.8)$$

where each x_u is an instance from X and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance $d(x_u, x)$ increases. Here k is a user-provided constant that specifies the number of kernel functions to be included. Even though $\hat{f}(x)$ is a global approximation to $f(x)$, the contribution from each of the $K_u(d(x_u, x))$ terms is localized to a region nearby the point x_u. It is common

to choose each function $K_u(d(x_u, x))$ to be a Gaussian function (see Table 5.4) centered at the point x_u with some variance σ_u^2.

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

We will restrict our discussion here to this common Gaussian kernel function. As shown by Hartman et al. (1990), the functional form of Equation (8.8) can approximate any function with arbitrarily small error, provided a sufficiently large number k of such Gaussian kernels and provided the width σ^2 of each kernel can be separately specified.

The function given by Equation (8.8) can be viewed as describing a two-layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values. An example radial basis function (RBF) network is illustrated in Figure 8.2.

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and σ_u^2 that define its kernel function $K_u(d(x_u, x))$. Second, the weights w_u are trained to maximize the fit of the network to the training data, using the global error criterion given by Equation (8.5). Because the kernel functions are held fixed during this second stage, the linear weight values w_u can be trained very efficiently.

Several alternative methods have been proposed for choosing an appropriate number of hidden units or, equivalently, kernel functions. One approach is to allocate a Gaussian kernel function for each training example $\langle x_i, f(x_i)\rangle$, centering this Gaussian at the point x_i. Each of these kernels may be assigned the same width σ^2. Given this approach, the RBF network learns a global approximation to the target function in which each training example $\langle x_i, f(x_i)\rangle$ can influence the value of $\hat{f}$ only in the neighborhood of x_i. One advantage of this choice of kernel functions is that it allows the RBF network to fit the training data exactly. That is, for any set of m training examples the weights $w_0 \ldots w_m$ for combining the m Gaussian kernel functions can be set so that $\hat{f}(x_i) = f(x_i)$ for each training example $\langle x_i, f(x_i)\rangle$.

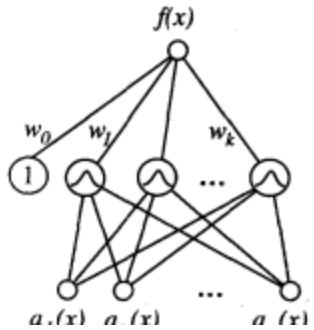


FIGURE 8.2
A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance x_u. Therefore, its activation will be close to zero unless the input x is near x_u. The output unit produces a linear combination of the hidden unit activations. Although the network shown here has just one output, multiple output units can also

To summarize, radial basis function networks provide a global approximation to the target function, represented by a linear combination of many local kernel functions. The value for any given kernel function is non-negligible only when the input x falls into the region defined by its particular center and width. Thus, the network can be viewed as a smooth linear combination of many local approximations to the target function. One key advantage to RBF networks is that they can be trained much more efficiently than feedforward networks trained with BACKPROPAGATION. This follows from the fact that the input layer and the output layer of an RBF are trained separately.