[This is a restored text of Part 1, comprising chapters 1 to 5, which I wrote, from a book on Machine Learning which was published in 1986. (Roy Rada authored Part 2.) I write "restored" because the production quality of the 1986 publication was really poor: it was riddled with misprints. We had corrected the bulk of these in proof, but, frustratingly, our proof corrections were ignored. Now a rectified version is available to the public -- 33 years too late! At any rate the text is much freer from misprints than the published version; but some figures are still missing. I intend to reconstruct and insert most of the missing figures over the coming months. As figures are reinserted, I will amend the release date below to indicate the latest revision.

Richard Forsyth. (Release date 13 March 2018.)

Please cite as:
Forsyth, R.S. (1986). Part 1, Machine learning for expert systems. In: Forsyth, R.S. & Rada, R. *Machine Learning: Applications in Expert Systems and Information Retrieval*. Chichester: Ellis Horwood. pp. 14-103.
]

Extract from the **Preface (**the only piece of writing by me that has been widely quoted):

> .... "Many people, other than the authors, contribute to the making of a book, from the first person who had the bright idea of alphabetic writing through the inventor of movable type to the lumberjacks who felled the trees that were pulped for its printing. It is not customary to acknowledge the trees themselves, though their commitment is total." ....

1

Introduction to machine learning

Machine learning is the key to machine intelligence just as human learning is the key to human intelligence.

In the natural world, species that have only rudimentary learning abilities are called 'primitive' -- insects, molluscs and worms, for instance. No one would dream of calling them intelligent. More advanced groups, such as Cetaceans and Primates, are characterized by a greater capacity to learn.

In the clinical world, there are case histories of patients with brain damage who lose the ability to remember new facts and events. If the condition develops in adulthood, the patient can still recall early memories and retains skills learned in childhood, but cannot modify his (or her) behaviour. Such people have to be institutionalized. They exhibit in exaggerated form a problem that is seen in senility -- not so much loss of memory as loss of adaptability.

Yet, in the field of Artificial Intelligence (AI), researchers have persisted in trying to build fully-fledged 'adult' systems which cannot learn for themselves at all. Early attempts to devise learning machines, during the cybernetic days of AI, proved disappointing, so the whole idea was dropped. Only recently has it been revived.

Our contention is that this revival of interest in machine learning is important, and will continue until systems capable of self-improvement become the norm rather than the exception. Indeed we believe that many AI problems (such as speech understanding) are so difficult that they can only be solved by systems that go through a 'childlike' phase. After all, we do not expect babies to emerge from the womb and ask politely for a drink of milk.

1.1 THE MEANING OF LEARNING

When a computer system improves its performance at a given task over time, without re-programming, it can be said to have learned something. We will accept automatic performance improvement with experience as a rough-and-ready definition of learning, without delving too deeply into the philosophical implications. Note that this implies a yardstick for measuring performance: if we cannot evaluate a system's performance, we cannot say whether it has learned anything.

In practice, learning algorithms attempt to achieve one or more of the following goals:

-- provide more accurate solutions;
-- cover a wider range of problems;
-- obtain answers more economically;
-- simplify codified knowledge.

The last goal presumes that simplification of stored knowledge is valuable for its own sake. For example, a system might re-arrange its knowledge base so that it was more intelligible to human readers. Even if its performance at the task was no better, this could well be useful.

However, it is the first two criteria (accuracy of solutions and range of applicability) which usually have the highest priority.

1.2 THE PHILOSOPHY OF INDUCTION

Before discussing how machines (which almost always means computers) may be made to learn, it is wise to see what philosophers and psychologists have said about the subject. We, too, may be able to learn from the past.

Philosophers have long been fascinated by the process of induction -- i.e. formulating general laws by examination of particular cases. Clearly induction is the foundation not merely for most of our day-to-day learning, but for the whole edifice of science as well. For this reason many philosophers have looked at the part played by inductive reasoning in scientific discovery. A typical act of induction goes something like this:

I have seen lots of white swans.
I have never seen a non-white swan.
==========
Therefore, all swans are white.

Another old favourite is the sunrise 'problem'.

Yesterday the sun rose in the East and set in the West.
Every day of my life it has risen in the East and set in the West.
Never in living memory has anyone seen it do anything else.

Throughout recorded history it has always risen in the East and set in the West.
==========
Therefore, it will do so tomorrow as well.

Those innocuous acts of commonsense inference are in fact logically invalid, and philosophers have spent many sleepless nights attempting to find rational grounds for validating them -- not because they expect the sun to rise in the West tomorrow, but because they would like to put such crucial conclusions on firmer footing. After all, Australian swans are in fact black.

Francis Bacon, John Stuart Mill, Bertrand Russell and, to a lesser extent, Ludwig Wittgenstein are the philosophers who have devoted most attention to the problems of induction. They have considered particularly its role in the scientific method.

Unfortunately, for our purposes, they have been less interested in how to do it than in how to justify it. Their objective was to frame rules governing inductive argument just as logicians, from Aristotle to Boole, have framed rules for deductive argument. As J. S. Mill said (in his System of Logic): "what induction is, and what conditions render it legitimate, cannot but be deemed the main question of the science of logic".

In this enterprise they have only been partly successful. Nevertheless, the AI practitioner who is chiefly interested in how to mechanize the process of induction can glean a number of hints from their work.

Bacon, for example (*First Book of Aphorisms*), stressed the importance of negative evidence, and the tendency of the human mind to overlook it. (See Hampshire, 1956.)

> "It is the peculiar and perpetual error of human intellect to be more excited by affirmatives than by negatives; whereas it ought properly to hold itself indifferently disposed towards both alike. Indeed in the establishment of any true axiom, the negative instance is the more forcible of the two."

He also pointed out that an inductive leap, to be of value, must go further than the facts warrant. When it does so, and is subsequently confirmed by observation, our confidence in it is strengthened.

> "But in establishing axioms by this kind of induction, we must also examine and try whether the axiom so established be framed to the measure of those particulars only from which it is derived, or whether it be larger and wider. And if it be larger and wider, we must observe whether by indicating to us new particulars it confirm that wideness and largeness as by a collateral security; that we may not either stick fast in things already known, or loosely grasp at shadows and abstract forms; not at things solid and realised in matter."

One final remark from Bacon, is perhaps his most apt: "the understanding must not therefore be supplied with wings, but rather hung with weights, to keep it from leaping and flying".

Two centuries after Bacon, Mill laid down four primary 'experimental methods' for inducing general laws from particular cases. These were:

-- the Method of Agreement
-- the Method of Differences
-- the Method of Residues
-- the Method of Concomitant Variation.

For example, the method of differences can be summed up as follows: take away factors affecting the situation one by one, two by two, and so on, to find the invariable and unconditional antecedent factors; then you have found the cause or causes of the phenomenon under investigation. (See Passmore, 1968; Russell, 1961.) The method of concomitant variation involves looking for factors that vary together, or in inverse proportions; for example, the height and momentum of a weight when it is dropped to the ground.

Mill's four methods, however, were specified before the computer age, and are hence too vague to serve as plans for a program designer.

Russell also wrestled with the problem of justifying inductive reasoning and eventually admitted defeat. As he says in The Problems of Philosophy, the inductive principle is "incapable of being proved by an appeal to experience". Its role in human thought, however, is so fundamental that "we must either accept the inductive principle on the grounds of its intrinsic evidence, or forgo all justification of our expectations about the future" (Russell, 1912). In other words, if you don't believe induction, you can't believe anything.

The inductive principle, as he saw it, was essentially probabilistic. In brief, when two things, such as smoke and fire, have been found to go together many times and never found apart, then 'a sufficient number of cases of association will make the probability of a fresh association nearly a certainty, and will make it approach certainty without limit'. No smoke without a fire, as they say.

Wittgenstein's contribution was to emphasize, or re-emphasize, simplicity. He asserted in the *Tractatus Logico-Philosophicus*, 6.363, that "the procedure of induction consists in accepting as true the simplest law that can be reconciled with our experiences" (Wittgenstein, 1961). Thus he recognized that many generalizations could be consistent with the evidence, and resurrected Occam's Razor as a means of choosing between them. For instance, having noticed that lightning flashes are invariably accompanied by loud bangs, it is simpler to assume that lightning causes thunder directly than that there is a heavy-metal rock group in a Jumbo jet who fly around looking for electrical storms so that they can re-charge their batteries and give their drummer a chance to practice his drum rolls with the amplifier at full blast.

The latter sort of explanation is difficult to disprove on the grounds of evidence alone, as the resilience of numerous superstitions testifies. Often such theories are only rejected because of internal contradictions. Indeed the history of science is littered with discredited hypotheses that were embellished with so many baroque flourishes that they collapsed under the weight of their own implausibility, even though they coincided with the known facts.

Thus, although philosophy has not laid the blueprint for an inductive engine, it does provide some guidelines for people wishing to build one.

-- Do not neglect negative evidence (Bacon).
-- Look for concomitant variation in the causal factors and the result (Mill).
-- The more cases of association observed, the more likely the association is to be generally true (Russell).
-- Prefer simple to complex generalizations (Wittgenstein).

This may seem no more than common sense, but then induction is a commonsense process which philosophers seek to clarify and AI workers to emulate.

1.3 THE PSYCHOLOGY OF LEARNING

Induction in science is a public procedure. In daily life, however, induction goes on in private whenever we learn from experience. As such, it has been extensively studied by psychologists for over a century.

Broadly speaking, psychological theories of learning fall into two groups. The S-R (Stimulus-Response) theorists regard the organism as a black box. They are interested in rules or formulae relating inputs (stimuli) with outputs (responses), but do not claim to model what is going on inside the animal's brain.

For example, the formula

```
P[n] = 1-(1-P[1]) * (1-theta) ^ (n-1)
```

can be used to predict the probability of a response on the nth trial, P[n], in a learning experiment where 0 is the probability of making a connection on any reinforced trial. This generates the familiar 'learning curve' depicted in Fig. 1.1. Its slope depends on the numeric value of theta. But the mathematical psychologists are interested purely in fitting behavioural data. They would deny that theta has the status of a mental construct. As far as they are concerned it is a parameter in an equation, and no more than that.

Cognitive theorists, on the other hand, do attempt to describe the mental structures which are constructed within the nervous system. Interestingly enough, they borrow extensively from computing concepts and terminology to describe what they think is going on in there.

S-R theorists speak of 'learning' and experiment mainly with rats and pigeons. Cognitive psychologists usually talk of 'memory' and do most of their experiments on human beings. At present cognitive theorizing is in the ascendant (partly at least because cognitive theories lend themselves to computer simulation), but the debate continues between adherents of the two approaches. It will be a very long time before a unified psychological theory of learning emerges that fits the multiplicity of experimental data concerning human and animal learning.

In the meantime the AI worker looking to psychology for ideas on how to build learning systems will be disappointed. To put it bluntly, the psychologists do not know how it is done. For many years behaviourism was the dominant orthodoxy in experimental psychology and behaviourists eschew mentalist concepts. The idea that a rat is forming and testing hypotheses when it runs a maze or that a pigeon is doing so as it pecks for food was anathema to behavioural scientists from about 1920 to 1970. It is hardly surprising, therefore, that psychologists cannot properly represent something they have only recently admitted to exist.

They can predict how rapidly (or slowly) people (or pigeons) will pick up various tricks in various situations. They can also tell you how fast they will be forgotten (on average). But the S-R theorists do not want to explain how it actually works, and the cognitive theorists are unable to do so.

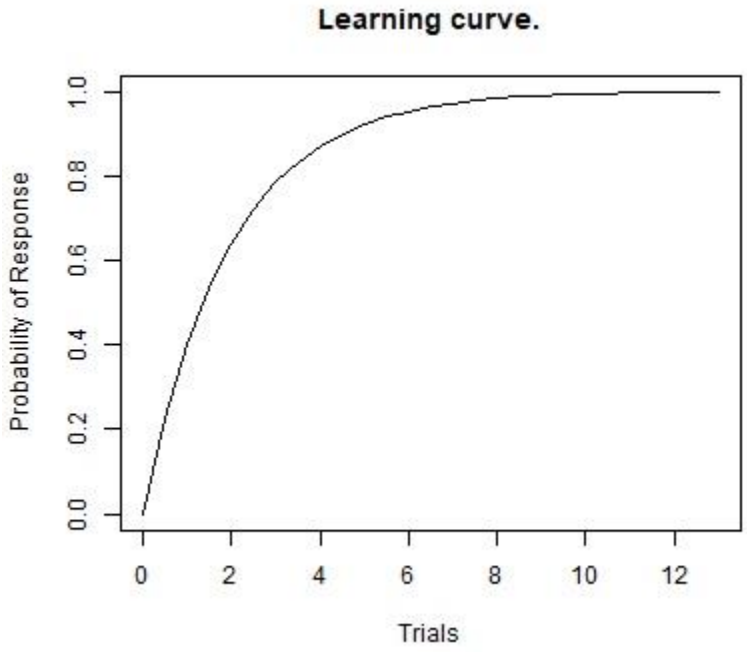

Fig. 1.1 -- Learning curve.

The formula

P[n] = 1-(1-P[1]) * (1-theta) ^ (n-1)

can be used to predict the probability of an animal making the correct response on trial n of a learning experiment. This may well match the form of observed results, but it does not attempt to explain how those results were generated. In particular the parameter theta does not have the status of a mental construct.

This is not quite so disgraceful as it seems at first glance. Part of the explanation lies in the disparity between neurons and notions. The modern psychologist who trains a rat to run a maze knows that it has developed some sort of 'cognitive map' of the maze. The physiologist who cuts up the poor creature afterwards knows a good deal about how its nervous system works. But nobody yet knows how to put those two levels of description together. How is the software (the cognitive map) implemented in terms of the hardware (the neural interconnections)?

Ask again in a decade or two!

Only two conclusions have emerged that would command universal assent among psychological researchers into learning. One is that short-term memory (STM) is distinct from long-term memory (LTM) and very much smaller: STM is generally reckoned to be limited to about 7 'chunks' of information while LTM is for practical purposes unlimited. The second is that feedback, or knowledge of results, is absolutely crucial to the acquisition of novel skills, and must be provided quickly. (See Hilgard & Bower, 1966; Bolles, 1979).

1.4 A FRAMEWORK FOR LEARNING

Having gathered what we can from the researches of philosophers and psychologists, we will spend the rest of this book investigating computer systems which model the process of learning in some way. In order to be able to compare them, we need a general framework for discussion.

Looked at from a long way away, all systems designed to modify and improve their performance share certain important common features. Fig.1.2 is a diagram of the four major components of a typical learning system. Essentially this sketches a pattern recognizer which learns to associate input descriptions with output categories, but as we shall see later many systems that are not overtly concerned with pattern recognition also fit into this general framework.

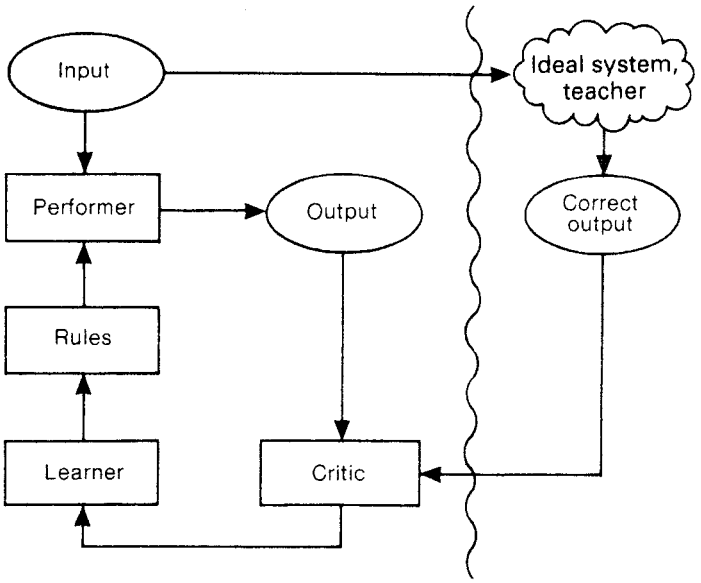

Fig. 1.2 -- A typical learning system.

This outline shows the logical layout of a typical learning system. To the left of the wavy line is the learning system, to the right is the ideal system or teacher, whose behaviour it is trying to match. The *Critic* compares the actual with the desired output and passes on feedback to the *Learner*. The learner attempts to modify the *Rules* (or knowledge base) to improve the system's performance. The *Performer* uses those rules to guide its performance at the task (but leaves the job of amending them to the learner module).

Note that the system contains a feedback loop. We will briefly describe its main components in turn, by going round this feedback loop.

The *Critic* compares the actual with the desired output. In order to do so, there must be an 'ideal system', as we call it, against which the system's behaviour is measured. In practice this may be a human expert, or teacher. For instance, if the task is medical diagnosis, the ideal may be the diagnosis given by a top consultant when faced with the patient whose history is being presented to the computer as input. The job of the critic is known as 'credit assignment' or alternatively 'blame assignment'. It must assess deviations from correct performance.

This can be simple or complex. In a simple case it might compare (for example) rainfall as forecast with actual rainfall. If 0.5mm of rain fell and the system predicted 1.9mm then it is only a matter of subtracting one number from another and passing the difference on as feedback to the learning module. In other circumstances there may be more work for the critic to do to ascertain what went wrong. For example, after losing a game of chess, it may not be at all obvious where the computer blundered.

The *Learner* is the heart of the system. This is the portion that has responsibility for amending the knowledge base to correct erroneous performance. A large number of learning strategies have been proposed, many of which we will examine in Chapters 2, 3 and 4.

The *Rules* are the data structures that encode the system's current level of expertise. They guide the activity of the performance module. The crucial point is that they can be amended. Instead of a Read-Only knowledge base (as in today's Expert Systems) the rules constitute a Programmable-Erasable knowledge base. Obviously they must only be modified under strictly defined conditions or chaos will result. Other forms of knowledge representation than condition-action rules have been used successfully, but we use the term 'rules' as a kind of shorthand for the moment.

Finally, the *Performer* is the part of the system that carries out the task. This uses the rules in some way to guide its activity. Thus when the rules are updated, the behaviour of the system as a whole changes (for the better, if all goes according to plan).

Two other terms need to be defined before we begin our examination of practical learning methods 'description language' and 'training set'.

The description language is the notation or formalism in which the knowledge of the system is expressed. There are two kinds of description language which are important. The first is the notation used to represent the input examples. One of the simplest of input formats is the feature vector. Each aspect of the input example is measured numerically, and the vector of measurements defines the input situation.

The second kind of description language is that chosen to represent the rules themselves. As we shall see in Chapter 3, the expressiveness of the description language in which the rules are formulated is critical to the success of any learning algorithm. It also has a bearing on how readily the knowledge can be understood, and hence on whether it can be transferred to people.

The notion of a 'training set' is important in understanding how a machine learning system is tested. Typically there is a database of examples for which the solutions are known. The system works through these instances and derives a rule or set of rules for associating input descriptions with output decisions (e.g. disease symptoms with diagnoses). But, as Bacon pointed out, rules must be tested on cases other than those from which they were derived. Therefore there should be another database, the test set, of the same kind but containing unseen data. If the rules also apply successfully to these fresh cases, our confidence in them is increased.

This preliminary definition of terms enables us to compare and contrast learning systems in a consistent fashion from now on. For example we can ask

> what description language does it employ?
> how does the critic evaluate performance?
> how does the performer make use of the rules?

and so on.

1.5 WHY BOTHER?

Why is machine learning worth striving for? The answers are almost as various as the applications of computers. Here are just a couple of possibilities:

> -- a computer program that starts off knowing only the rules of the ancient oriental game of Go, and ends up defeating the world champion;

-- a system that scans meteorological records of the past few decades and works out how to forecast the weather more reliably than at present.

In practice, the motivation for the resurgence of interest in machine learning has come from the new discipline of 'knowledge engineering'. Knowledge engineers are people who build expert systems, and their job is not easy. Traditionally (if one can speak of tradition in so new a field) they have had to work long and hard with a 'domain specialist', a human expert, to codify the expert's knowledge in symbolic form -- e.g. as condition-action rules.

Experts are notoriously bad at formalizing their expertise -- even if they are not worried about revealing trade secrets. The knowledge engineer must coax out of them know-how they are scarcely aware of possessing. Often this means building a flawed prototype system, submitting it to the expert's disparaging criticism, revising it, doing the same again, and repeating the cycle until the system reaches an acceptable level of performance.

This arduous process has come to be known as the 'knowledge acquisition bottleneck'. Machine learning offers one way through that bottleneck. Ideally a learning system could take a data base of example cases and come up with a set of rules for doing the expert's job. Even if completely automated knowledge acquisition of this kind proves too ambitious, the ability to refine a partial or inconsistent knowledge base derived from a human would be valuable.

And, of course, machine learning opens up the exciting possibility of synthesizing totally new knowledge -- discovering concepts and patterns that humans have never even thought of.

1.6 REFERENCES

Bolles, Robert (1979) *Learning Theory*: Holt, Rinehart & Winston, New York.
Hampshire, Stuart (1956) ed. *The Age of Reason*: Mentor Books, New York.
Hilgard, E. R. & Bower, G. H. (1966) *Theories of Learning*: Appleton Century-Crofts, New York.
Passmore, John (1968) *A Hundred Years of Philosophy*: Penguin Books, Middlesex.
Russell, Bertrand (1912) *The Problems of Philosophy*: Oxford University Press.
Russell, Bertrand (1961) *History of Western Philosophy*: Allen & Unwin, London.
Wittgenstein, Ludwig (1961) *Tractatus Logico-Philosophicus* tr. by Pears & McGuinness: Routledge & Kegan Paul, London.

2

Black box methods

One way of trying to understand a complex phenomenon is to ignore its internal structure and treat it as a 'black box'. This notion has proved useful in a number of disciplines ranging from engineering to biology. The point about a black box is that no one really cares about what goes on inside it. Only the inputs and outputs of the system under scrutiny (which may be a natural phenomenon, a living organism or a man-made artefact) are studied. As long as the relationships between inputs and outputs can be specified precisely, it is not important how they are achieved (Forsyth & Naylor, 1985).

A black box is completely specified by its input-output behaviour. It does not matter whether that behaviour is realized electronically, hydraulically, mechanically, or by wet meat.

This methodology has obvious affinities with the S-R approach to the psychology of learning, as outlined in Chapter 1. Many behavioural scientists favour a black-box approach to learning because it is very hard to decide what is going on inside the brain of an animal (still less its mind) as it learns.

Some investigators in the field of machine learning have followed this lead. In this chapter we survey a number of black-box learning systems. They all share two distinguishing features:

(1) a mathematical bias;
(2) a 'write-only' description language.

The mathematical bias means that they tend to employ well-established procedures from the realms of statistics and control theory. Designers of such systems think nothing of multiplying or inverting matrices, calculating eigenvectors and so forth. These systems frequently require considerable processing power for heavy number-crunching.

Partly as a consequence of this, the knowledge that the system gains during its training phase is opaque. It may calculate a covariance matrix or optimize a set of coefficients; but even a mathematically sophisticated person cannot inspect knowledge in this format and readily determine what the system has learned. This is what we mean by saying that black-box learning systems have a 'write-only' knowledge base. It is computable, but not intelligible. Being able to look inside the black box is not one of the design goals.

In contrast, developers of structural learning systems tend to follow the footsteps of the cognitive psychologists. Their systems are intended to generate knowledge that is humanly comprehensible. But we postpone detailed consideration of such systems until Chapter 3.

2.1 INTRODUCTION TO PATTERN RECOGNITION

This book is concerned with all kinds of machine learning, but especially those which can be harnessed for the development of expert systems. Black-box learning systems, however, have almost all been applied in the area of pattern recognition.

The two fields, pattern recognition and expert systems, have different historical roots, different learned journals and, above all, different practitioners (very often in different academic

departments). They belong to different subcultures, so it is worth spending a few paragraphs pointing out the similarities in their aims and methods. In particular, it is worth stressing that pattern recognition lies at the heart of all expertise, whether human or computerized.

Here are some typical pattern recognition tasks:

equipment fault diagnosis;
medical diagnosis;
fingerprint identification ;
voice recognition.

In all pattern recognition tasks there are a number of measurements made of an event or object. These raw measurements are transformed in some way into a set of features, and the features are used by a decision procedure to assign the event to one category or another. Thus the typical pattern recognition system is a classifier. This is illustrated in Fig. 2.1.

Clearly, classification is a process of information reduction. The central operation (feature extraction) is logically unnecessary; but all practical systems have this intermediate step. Coupling the decision algorithm directly to the raw data is computationally infeasible. In fact the choice of features is crucial to the success of any classifier.

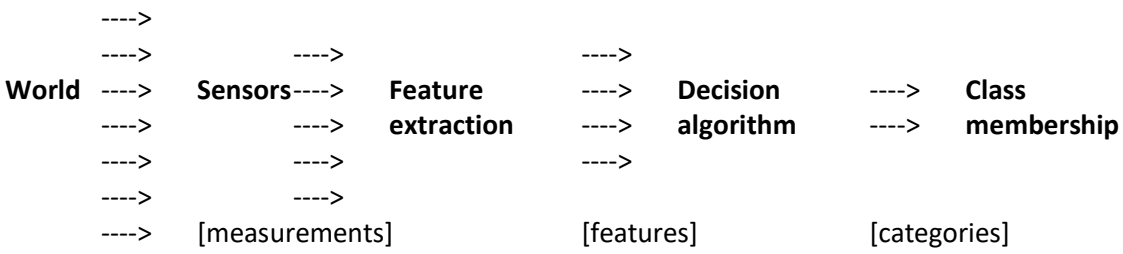

Figure 2.1 -- Pattern recognition.

Pattern classification is a data-reduction process. Normally three stages are involved. First the attributes of the outside world are sampled or measured in some way. Next these raw measurements are processed to give a smaller number of features. Finally the features are examined and the system picks one of a small number of classes and assigns the pattern to that class. The problem is the many-to-one mapping of inputs to outputs.

Let us take birdsong recognition as an example. Suppose a system is required to distinguish nightingales from thrushes, by listening to their songs. It will have to go through the following three stages.

(1) Measurement. Birdsong is recorded via a microphone and pressure waves are transformed into a time-varying amplitude signal. (See Fig. 2.2.)
(2) Feature Extraction. This electrical signal is converted into a vector of parameters giving (for instance) number of zero crossings, maximum energy level, total duration of silence, number of detectable silences, average energy in the 10-20 KHz bandwidth, and much else besides.
(3) Classification. The feature vector is processed mathematically and a decision algorithm applied to decide category membership.

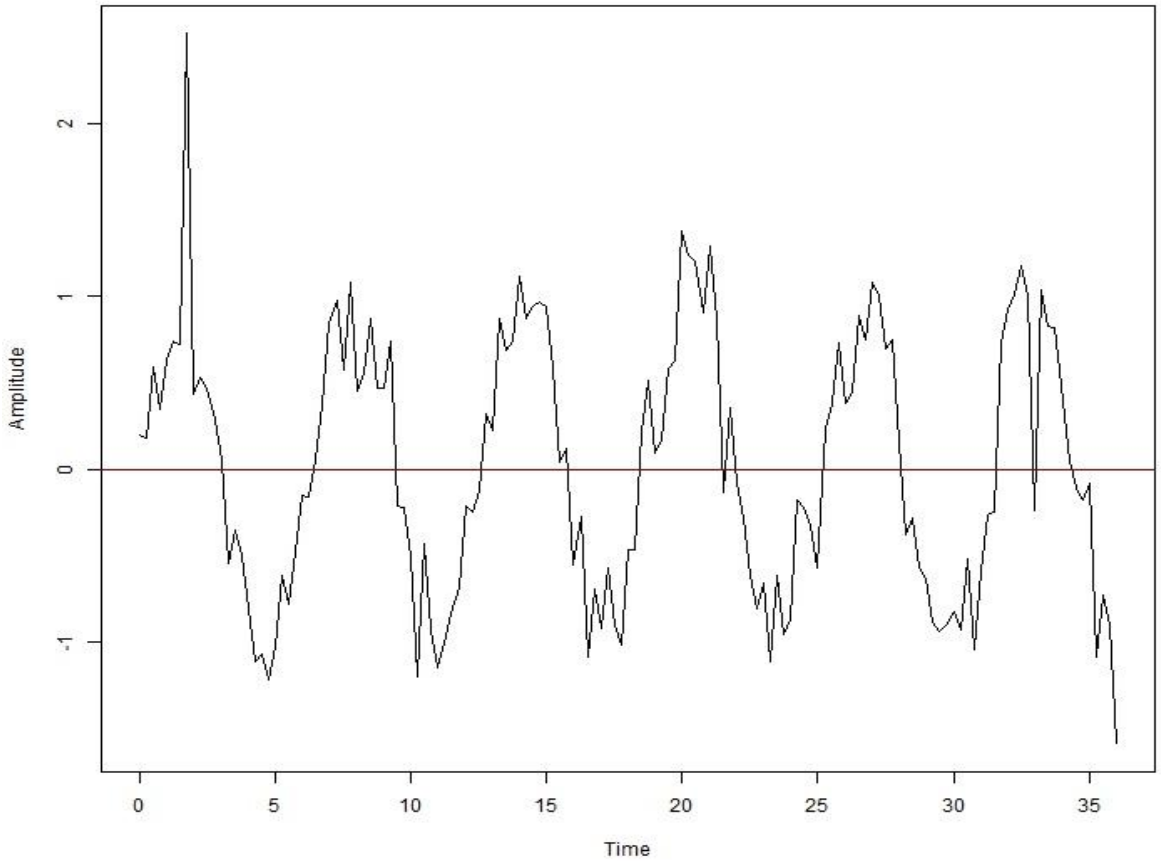

Figure 2.2 -- A typical acoustic waveform.

A microphone converts waves of air pressure into voltage, varying with time. The raw temporal signal is difficult to process as it stands, so it is usually transformed further into a vector of numeric features. One commonly used feature is the number of zero-crossings. This is an index of the overall frequency of the sound: the more times the wave form crosses the zero line, the higher the frequency.

None of these stages is particularly simple. In particular, an informed choice of features at stage 2 can make all the difference. This is where human expertise is most needed at the design stage.

Before carrying out this three-step classification process, the system must be trained. In the training phase the decision algorithm is optimized, in some sense, for the discriminations it has to perform.

Note that although learning is commonly used for tuning the decision algorithm, it is only rarely used for the feature extraction process. There are examples of systems that pick useful features from a large set of potentially useful ones; but there are very few that make up their own features from scratch.

After this detour into the field of pattern recognition, ret us return to expert systems. Here is a list of some expert system applications:

equipment fault diagnosis;
medical diagnosis;

fingerprint identification;
voice recognition.

It is identical to the pattern recognition tasks that we listed earlier (deliberately, of course). This overlap is not complete; but expert systems have to recognize patterns, and once they have done so, the rest of their task is often trivial. For example, in a medical consultation, the difficult part might be to tell whether the patient is suffering from gastric ulcer or stomach cancer. Once that diagnostic problem is solved, the choice of therapy is highly constrained. It may simply be a matter of looking up the prescribed therapy in a table.

Expert systems, therefore, need to recognize patterns. From now on we will treat systems that learn to recognize patterns as directly relevant to our quest for self-improving expert systems.

The systems described in this chapter use a description language in which input patterns are presented as feature vectors, always numeric and sometimes binary. Thus an input example is a vector of numbers. If there are F features, then this vector defines a point in F-dimensional space. Various mathematical/geometric terms are used to describe regions in this abstract space. Many of the methods attempt to partition the feature space so that clusters of examples of one kind are together and clusters of another kind in a different region. It is quite easy to imagine examples in 2D or 3D feature space (e.g. with up to three features, such as age, weight and height); but the terminology for multi-dimensional feature space can become confusing, so we list the correspondences in Table 2.1.

Table 2.1

2D	**3D**	**Multi-D**
space	3-space	hyperspace
area	volume	volume
curve	surface	hypersurface
line	plane	hyperplane
circle	sphere	hypersphere
square	cube	hypercube
polygon	polyhedron	polyhedron

2.2 BLACK BOXES AND GREY MATTER

The brain really is a black box (at present); but the digital computer is not. At its darkest it is dirty grey: it is always possible, in principle, to look inside. In practice, however, it may not be desirable or profitable to do so.

However, from our vantage point, we can peep into the box and say that learning systems of this type fall into two main groups, on the basis of how they store their knowledge. There are those that adjust the parameters or coefficients of a discriminant function until it is optimal, or at least satisfactory, according to predefined criteria; and there are those which perform what amounts to an indexing operation. Borrowing the terminology of Samuel (1967), we say that systems of the second type construct 'signature tables'.

We can illustrate the difference with a simplified example. Suppose the objective is to classify military aircraft as fighters or bombers. There are two features: maximum speed (in km/h) and maximum load at take-off (kg). Table 2.2 gives two examples of each class.

Table 2.2

Fighter	Speed	Load	Bomber	Speed	Load
F-15C	2443	20185	Blackjack-A	2200	267000
SU-27	2445	28800	TU-22M	2036	118000

There would be many more examples than this in a realistic training set.

A parameter adjustment system would typically look for weightings by which to multiply the features S (speed) and L (loading) in a function such as

$$F = w[0] + w[1]*S + w[2]*L$$

where $F > 0$ would be taken to indicate that the aeroplane was a fighter and $F < 0$ would indicate that it was a bomber. (This is a simple linear discriminant function.)

Since the bombers are heavier and (on the whole) slower, the final weightings might be: $w[0] = 640$, $w[1] = 1$, $w[2] = -0.0566$. This would give the discriminant function

$$F = 640.00 + 1.00*S - 0.0566*L$$

where a positive result indicates a fighter, as above.

A system based on signature tables, on the other hand, would attempt to quantize the feature range by establishing cut-off points or thresholds, giving S and L a small number of levels, and come up with a decision table such as Table2.3.

Table 2.3

	L < 22000 kg	**L intermediate**	**L > 50000 kg**
S > 2400 k/h	Fighter	Fighter	Bomber
S intermediate	Fighter	Bomber	Bomber
S < 1000 k/h	Bomber	Bomber	Bomber

The two variables have been divided at two threshold points into three levels each, giving a table with nine entries altogether. In fact the entries in this table might well be probabilities (based on frequencies of fighters and bombers in the training data) for each class, rather than just class labels as shown here.

The relationship between these two different kinds of knowledge representation is depicted in Fig. 2.3. It is evident from the diagram that the parameter-estimation approach involves creating a function to define the boundary between the two classes, while the signature-table approach produces a jagged or zigzag decision line. (Both methods can be elaborated greatly compared to the examples presented here.)

The two methods have slightly different strengths and weaknesses. One advantage of signature tables over functional parameters is that they are somewhat more transparent: the box is a lighter shade of grey. Another advantage is that tables can be used to partition the feature space in more diverse ways than linear functions. In general, however, a decision function can be tabulated and a signature table can be functionalized if the need arises.

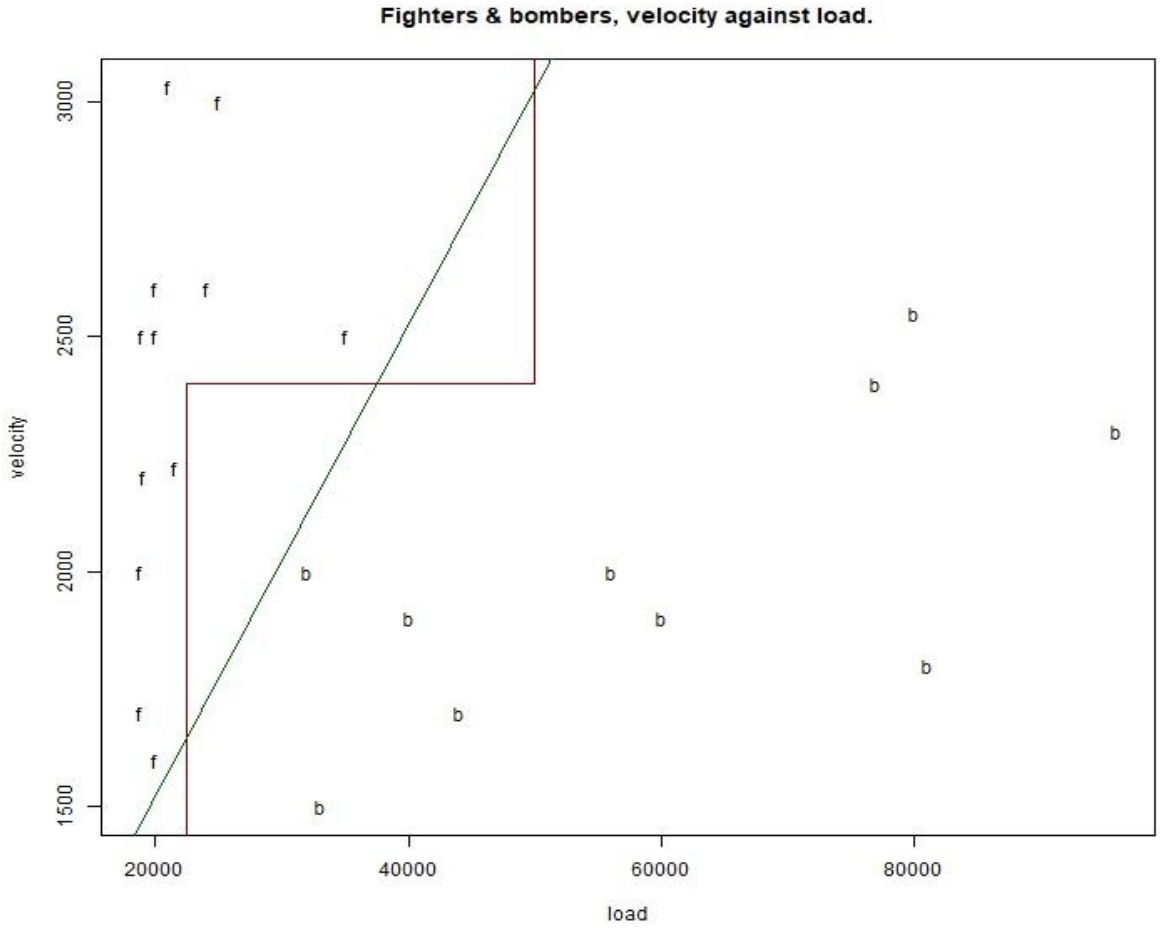

Figure 2.3 -- Parameter adjustment versus Signature Tables.

With only two features the objects in the training set can be displayed as points on a graph whose positions are fixed by plotting speed against load. Dots mark fighter planes (F) and crosses mark bombers (B). To distinguish the two classes a parameter adjustment (PA) system fits the parameters of a function. In this example, the steeply sloping diagonal divides F from B, though it is not always a straight line that is used. A signature table method (ST) creates, in effect, a jagged boundary, as shown here by the stepped line. The axes are divided at threshold points.

2.3 PARAMETER ADJUSTMENT

Parameter adjustment (PA) is one of the simplest forms of learning. We shall look briefly at four different systems which, between them, exemplify the most important parameter-adjustment techniques.

2.3.1 The Perceptron

One of the earliest, and still one of the best known, PA systems was the Perceptron (Rosenblatt, 1958, 1962). Originally it was put forward as a simple neurological model, but it is perhaps best regarded as one of a family of trainable classifiers with certain interesting properties. The device which Rosenblatt came to call the Mark I Perceptron has received most attention in the literature. This is the version we shall describe here. Its operation is sketched diagrammatically in figure 2.4.

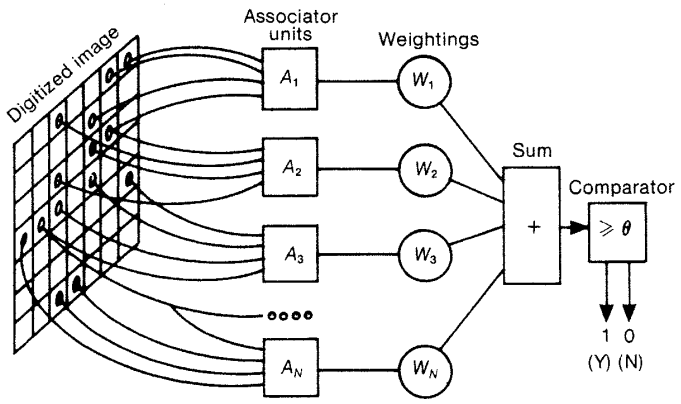

Figure 2.4 -- Mark I Perceptron.

> Outputs from the Associator Units are 1s or 0s computed by adding the inputs and comparing to a threshold. In other words, each Associator Unit (or feature detector) is a Perceptron in miniature. These outputs are multiplied by the Weightings then totalled and compared to a threshold theta (∑A[i]*W[i] >= theta). If the sum is greater or equal the system says Yes (1); if not, it says No (0).

In the first place an image is projected onto the picture plane and digitized as a grid of zeroes and ones, with 1 representing light and 0 dark. This is a crude approximation to the retina. The next level consists of a number of 'Associator Units'. Each of these receives inputs from several pixels in the digitized image. The associator units are the feature-detectors: they are threshold devices which produce binary outputs (rather like miniature Perceptrons themselves). For example, unit A7 may sample six randomly distributed pixels and 'fire' when four or more are lit. Firing just means transmitting a 1, rather than a 0, to the next level. Thus associator units are predicates which detect subpatterns in the input by local computations.

The outputs from the associator units are passed on to weighting units. Each binary value is simply multiplied by a numeric weighting. These weighted values are added up and compared to a threshold by the comparator. If the sum equals or exceeds the threshold, the system says Yes (pattern observed), otherwise it says No (pattern absent).

The only things that vary are the weightings. The Perceptron uses an error-correction learning algorithm in which the weightings after an erroneous response are adjusted as follows

```
w[j] := w[j] - a[j] * d
```

where d = 1 if the system said Yes and the teacher said No, and d = -1 if the system said no and the teacher said Yes.

In other words, no changes are made after a correct response, but after a mistake all the feature values (a[j]) are either added to or subtracted from the weightings (w[j]), depending on whether the system's output was too high or too low on the previous trial.

This is known as the proportional increment training strategy (Sklansky & Wassel, 1981). It can be proved to converge on an optimal set of weightings only when the two pattern classes are linearly separable -- i.e. when a linear function can separate the two groups in feature space. In many practical problems the classes are not linearly separable, although even in these cases the Perceptron may still give acceptable results.

There are many variations on the Perceptron theme. We will consider just one, in which it is given two minor extensions. First we extend the number of pattern classes beyond two. Secondly we allow features to have unrestricted numeric values, not just zero and one. The knowledge base for such an augmented Perceptron is simply a numeric matrix, as below.

Figure 2.5 -- Weighting matrix.

Output categories

Input features	C1	C2	C3	C4	C5	C6
F1						
F2						
F3						
F4						
F5						
F6						
F7						
F8						
	| V	| V	| V	| V	| V	| V

Column totals

In this augmented Perceptron, there are as many columns as there are categories in the classification task. When the input features are presented, they are multiplied by each column in turn. The total output for each column is added up and the largest total picked as the system's output. The error-correction algorithm subtracts the F values from the corresponding C values for the column that gave the wrong answer, and adds the F values to the column that failed to give the right answer.

In this example we have eight input features F1 to F8 and six pattern classes C1 to C6. In each cell of the array is stored the weighting for the contribution of the feature in that row to the category in that column. In operation the features F1 to F8 are presented as input. For each column, the feature values are multiplied by the appropriate weights, and the column totals added up. The column with the highest total is chosen as the output class.

When the system makes a mistake, all the F values are subtracted from the weightings in the column that gave the incorrect response, and the F values are added to the column that should have given the response (but failed to do so). Other columns are left unaltered. Weights that are too large are thereby reduced and those that are too small are increased.

In this form the Perceptron can be taught to perform a variety of useful classification tasks, but it can still only be guaranteed to work when the classes are linearly separable. This is because any weighted sum is a linear function. Few interesting problems involve linearly separable classes. This and other deficiencies of the Perception were pointed out by Minsky & Papert (1969).

2.3.2 Pandemonium

Another pioneering pattern recognizer was the picturesquely named Pandemonium -- a model put forward by Selfridge (1959) as a 'paradigm for learning'. Pandemonium learns in the sense that, having been given some examples of patterns and told what they are, it can then "guess correctly which pattern has just been presented before we inform it".

Selfridge describes the architecture of Pandemonium in terms of four levels of 'demons'. At the lowest level, data demons supply coded versions of the physical input -- typically a digitized picture. At the next stage the computation demons respond to various elementary features in the input and pass their output up in turn to a number of cognitive demons -- each one responsible for one identifiable pattern in the set of possible responses. Their job is to 'shriek' at a single decision demon, with a loudness proportional to the degree to which their pattern matches the features on display. The decision demon picks the loudest as the correct response.

Pandemonium "contains the seeds of self-improvement" in two ways. It can alter the strength of the connections between computation demons and cognitive demons; and it can alter computation demons. New ones are obtained by mutating old demons. This may mean changing some parameters of a subdemon more of less at random, or creating one which is the logical combination of two existing ones. (See also chapter 4.) Cognitive demons are specified by the task, and do not change.

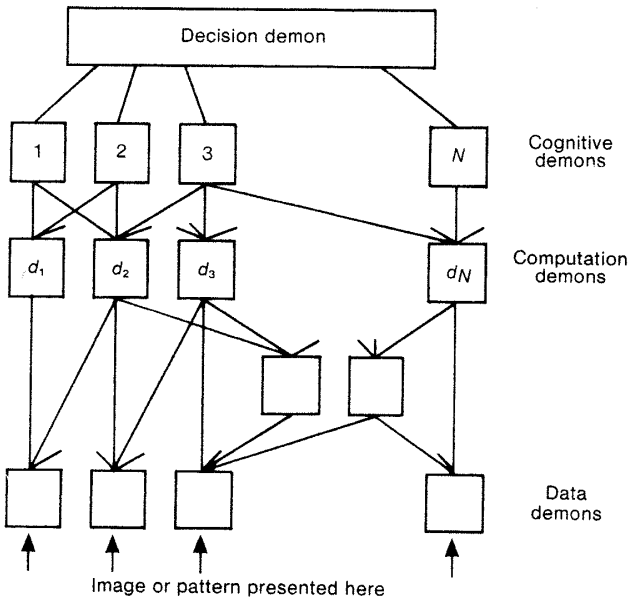

Figure 2.6 -- Pandemonium.

Computation Demons perform calculations on Data Demons to extract features. Cognitive Demons are specified by the task and do not change, but Computation Demons can be mutated according to their performance.

Selfridge & Neisser (1960) reported that programs based on the Pandemonium concept were successful at transliterating manually keyed Morse code and in recognizing hand-printed letters of the alphabet. Uhr & Vossler (1961) took this approach one stage further by developing a program that generated its own operators (i.e. demons). An operator, in their terminology, is a 5 x 5 submatrix, set to respond to a particular pattern of zeros and ones, that scans a larger 20 x 20 image

matrix and records the number of matches found, and their average horizontal, vertical and central position. Further operators then combine characteristics.

The resulting characteristics are then compared with lists in memory representing patterns, and the most similar chosen. The program gets better by varying amplifiers associated with each of the operators; by changing the lists in memory; and by creating or discarding operators. Novel operators are generated either as random 5 x 5 configurations or, more interestingly, as submatrices actually found in the large pattern. They are checked against stored operators before adoption, to avoid duplication.

This program has been tested against human performance in an experiment (Uhr, 1963). Subjects were required to learn to classify five variants of five types of 'meaningless' forms. The program consistently outperformed the people; but the program's authors commented that this superiority "should not be taken too seriously".

Both Selfridge and Uhr gave their models neurological interpretations (Uhr, 1966); but they are not very good neural models. They are mainly interesting as pioneering examples of machine learning, although (like the Perceptron) neither of them is capable of learning, for instance, whether an input field contains an odd or even number of figures of a kind that it can recognize in isolation.

Pandemonium, though it attracted less attention, was actually more powerful than the Mark I Perceptron. Its main advance was that it could generate new demons, whereas the Perceptron's associator units are fixed.

2.3.3 Hypersphere classifiers

Another kind of PA system is the hypersphere classifier described by Batchelor (1974). This is a generalization of an earlier technique known as nearest-neighbour classification (NNC).

In the NNC method a number of training examples are stored as representative points in the feature space. The feature vector of each example defines a point in a multi-dimensional space with as many dimensions as there are features. When a new case is presented to the classifier, it measures its distance from each of the known examples and assigns it to the same class as the nearest one. For this purpose, a distance measure must be defined. Usually the system minimizes the sum of squared deviations, and then uses a straight-line or Euclidean distance metric; but other measures are occasionally employed. A diagram plotting example bombers and fighters on a 2D feature space shows how this method works (Fig. 2.7).

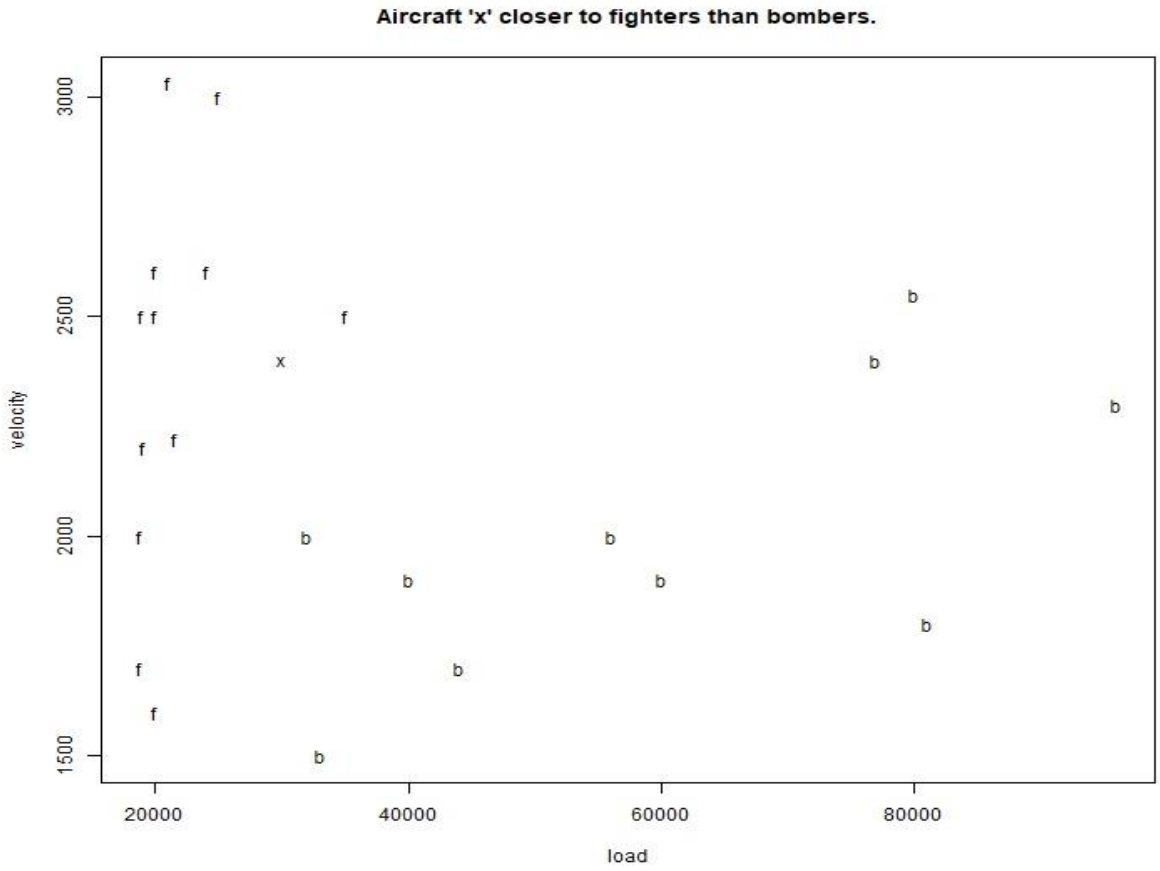

Figure 2.7 -- Nearest-neighbour classification.

Nearest-neighbour classification (NNC) works by measuring the distance of a given point in the feature space to the nearest point of known class, and assigning the unknown point to that class. Here the objects are aircraft measured by maximum speed and fully-laden weight. Letter b represents bombers (group B) and f's represent fighters (group F). The unidentified flying object indicated by the x would be classified as a fighter (group F) because its nearest neighbour is a fighter plane.

NNC systems suffer two problems which the hypersphere method attempts to overcome. First, they require a great deal of storage, since all or most of the training instances need to be stored. Secondly they are sensitive to 'rogue' or 'quirky' exemplars. Various statistical techniques have been proposed to remove outliers, though none is wholly satisfactory.

The hypersphere method tries to get round these problems by storing only prototypical exemplars called 'locates', which represent clusters of observed cases. In fact the precise feature values of the locates may never have occurred in any single training example.

The training procedure works as follows.

(1) Initialize the classifier with a few (i.e. one or two) locates.
(2) Optimize the positions and sizes of the hyperspheres surrounding each locate, using the training data one by one with the error-correction procedure described below.
(3) Test the classifier's performance. 3a. If the error rate is sufficiently low, halt. 3b. If it is unsatisfactory, add a new hypersphere.
(4) Repeat from step 2, unless there are enough hyperspheres.

The error-correction procedure adjusts the parameters of one or more hyperspheres. There are four distinct outcomes after each test case, where M stands for the machine's response and T for the teacher's response, either of which can be +1 or -1.

(A) M = T. No modification.
(B) M = -1; T = 1. (Latest point X lies outside all hyperspheres). The hypersphere closest to X is found, moved slightly closer to X and enlarged.
(C) M = +1; T = -1. (Latest point X is in the wrong hypersphere.) The hypersphere enclosing X is moved away from it and reduced in size.
(D) M = +1; T = -1. (X is in more than one hypersphere.) All the hyperspheres enclosing X are moved away from it and reduced in size.
Note that C is a special case of D.

To understand this process you should imagine the locates as the centres of multi-dimensional bubbles in the feature space. They move around, growing and shrinking, until nearly all the positive training examples are within their radius but hardly any of the negative ones are so enclosed. With two or three dimensions this is easy to visualize. With more dimensions it is harder, but the mathematics are essentially the same.

Moving a hypersphere means adjusting the coordinates of its locate, i.e. its feature values. Growing or shrinking one means altering an additional parameter, which gives its radius of influence.

The outer loop of the procedure starts with a single hypersphere (or only a few) and adds new ones until there is no gain from doing so. Batchelor suggests some heuristics for deciding where to place a new hypersphere. One idea is to take the least satisfactory hypersphere and split it in two. Another is to look for misclassified examples and take their centroid, or average on all dimensions.

In any case this incremental process can sometimes create degenerate hyperspheres, which have near-zero radius or which are completely enclosed by another one. So it is advisable to append a pruning phase afterwards which removes redundant hyperspheres until performance begins to deteriorate.

Batchelor gives various hints on the practical implementation of hypersphere classifiers.

(1) Put the initial (single) locate at the centroid of the positive training instances.
(2) Make the radius of the first hypersphere 'small'.
(3) Do not generate more than N/10*(F+1) locates, where N is the number of training samples and F is the number of features or dimensions.
(4) Use approximately 100/E training patterns, where E is the desired percentage error rate.

These rules of thumb do not, however, address the problem of deciding whether to classify positive or negative instances. This sort of classifier is asymmetrical; so it can make a difference which way round the two classes are assigned. For instance, Bomber = +1 and Fighter = -1 might give significantly better results than Fighter = +1 and Bomber = -1 (for the sake of argument). This question can only be answered by experiment, which implies that such systems have to be run twice to attain optimum performance.

Another disadvantage is that the extension to multiple classes is not a trivial matter. In fact such systems become impractical with a large number of pattern categories. The printable set of 96 ASCII characters would be far too large a domain.

However, with a small number of pattern classes (less than a dozen), hypersphere classifiers will generally outperform simple Perceptrons or Pandemonium-type systems. This is because they can adapt to much more diverse distributions within the feature space than systems based on linear functions. The hypersphere, naturally, is a non-linear decision surface.

2.3.4 The Boltzmann Machine

After Minsky & Papert's book on Perceptrons (1969), pattern recognizers based on overtly neurological principles virtually disappeared from the AI literature. Research on pattern recognition continued in other fields, such as applied mathematics and engineering, but without the neurological overtones.

Nevertheless, the fact that the brain is a trainable pattern recognizer cannot be disputed, and recently AI researchers have started to re-examine the idea of simulating neural networks. Why should they think it might be worth resurrecting the old 'discredited' notions of Rosenblatt, Selfridge and others?

First, there has been some progress in neurophysiology during the intervening 20 years. Second, and more important, there has been extraordinary progress in micro-electronics during the same period. Fabricating machines that simulate the brain is beginning to seem feasible.

A team at Carnegie-Mellon University, led by Geoff Hinton (1985), has designed a system to do just that. They call it the Boltzmann Machine, in honour of Ludwig Boltzmann, one of the founders of statistical mechanics. (We shall see later why his name is appropriate.) Theirs is a deliberate attempt to mimic the behaviour of neural networks, though admittedly their model is highly idealized.

A Boltzmann Machine is composed of a network of multiple computing elements, all of which work in parallel. It can be simulated on a conventional computer, but the ultimate aim is to use it as a blueprint for a novel form of computer architecture.

The Boltzmann Machine learns by trying, in effect, to construct internal representations of the input-output relationships it encounters in its environment. Its model of the world is represented by the strengths of association between elements, which are not meant for public scrutiny. It is a very dark box.

The elements in the network are threshold units, which produce a binary output (0 or 1) by adding up their inputs and 'firing' if the sum exceeds a certain quantity. Thus each processing element is a simplified, abstract neuron. The important point is that the elements react probabilistically. The threshold can be said to waver: the bigger the input, the more likely the unit is to fire, but the same input does not invariably trigger the same response. (See Fig. 2.8.)

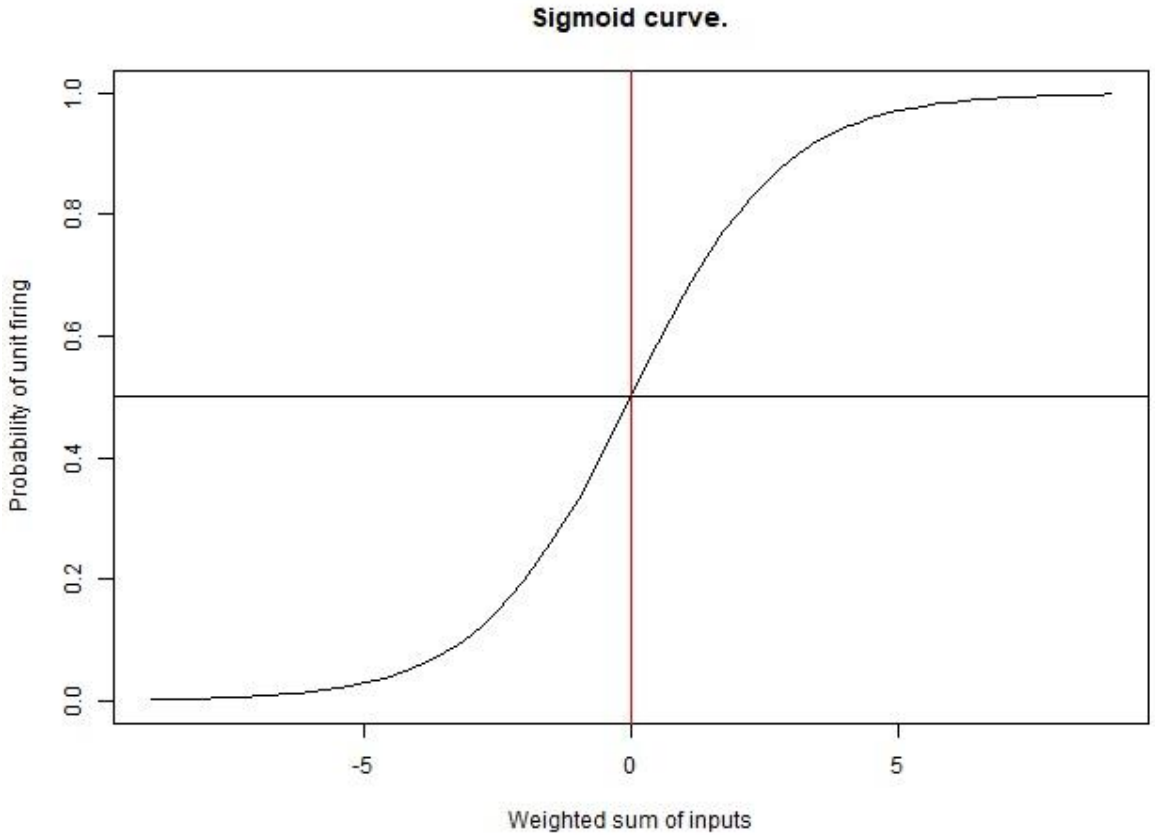

Figure 2.8 -- Probability of processing unit firing.

This graph shows the probability that a given element will fire as a function of the weighted sum of its inputs. The whole network can be made more deterministic by making this curve steeper, and it can be made less deterministic by making the curve flatter. The simulated annealing process begins with all the units having a flat curve and gradually makes them all steeper.

All processing units in a Boltzmann Machine are of the same type; and they all have the same threshold, zero. But the connection strengths between them may vary. Connection strengths are symmetrical, so that if the link in one direction (X to Y) has a weighting of 7.5 then so does the link in the opposite direction (Y to X). (This is not like the real nervous system.)

The network of interconnections can be far more complex than that of a Perceptron. There are input units from the outside world which receive the feature vector (coded as a string of zeroes and ones), and output units which, in effect, give the system's decision. In between, all kinds of links are permitted. There may be backward links on many levels, with feedback loops and so on, as illustrated diagrammatically in Fig. 2.9.

The system can be made to learn input-output relationships (i.e. to recognize patterns) by adjusting the interconnection weights in the following manner.

Phase 1

1(a) Clamp the training pattern to the input units, and the desired response pattern to the output units.

1(b) Allow the network to settle down to equilibrium.

1(c) Increment the weights between any two elements (by a small amount, delta) whenever they are both active together.

Phase 2
2(a) Remove the output connections (i.e. take away the teacher), but leave the inputs connected.
2(b) Let the network stabilize again.
2(c) Decrement the weights between any two elements which are simultaneously active, by delta.

The two phases are repeated alternately for as many iterations as are required to give satisfactory input-output behaviour in phase 2 (the unsupervised response).

[....]
Figure 2.9 -- Boltzmann Machine network layout.

> The connection network in a Boltzmann Machine can be much more complicated than that of a Perceptron. There are typically many layers, and links may exist between different layers. The 'black box' is enclosed in the rectangle.

This method has been proved to minimize the discrepancy between the structure of the network's internal model and the actual structure of the input-output relationship. Thus the system learns to give the same responses in phase 2 that it has been taught in phase 1.

Steps 1(b) and 2(b) are both instances of a procedure known as 'simulated annealing', by analogy with the hardening of metal as it cools. When a substance is hot, there is plenty of essentially random motion among the molecules that compose it. As it cools, they settle into fixed positions.

Likewise in steps 1(b) and 2(b), the input-output relationship of all the units in the network starts off with a greater degree of randomness and becomes increasingly deterministic. This analogy with temperature reduction is intended to avoid a well-known pitfall in machine learning -- the problem of local optima.

To simplify matters, for the time being, imagine a ball bearing on a jagged surface, as outlined in Fig. 2.10.

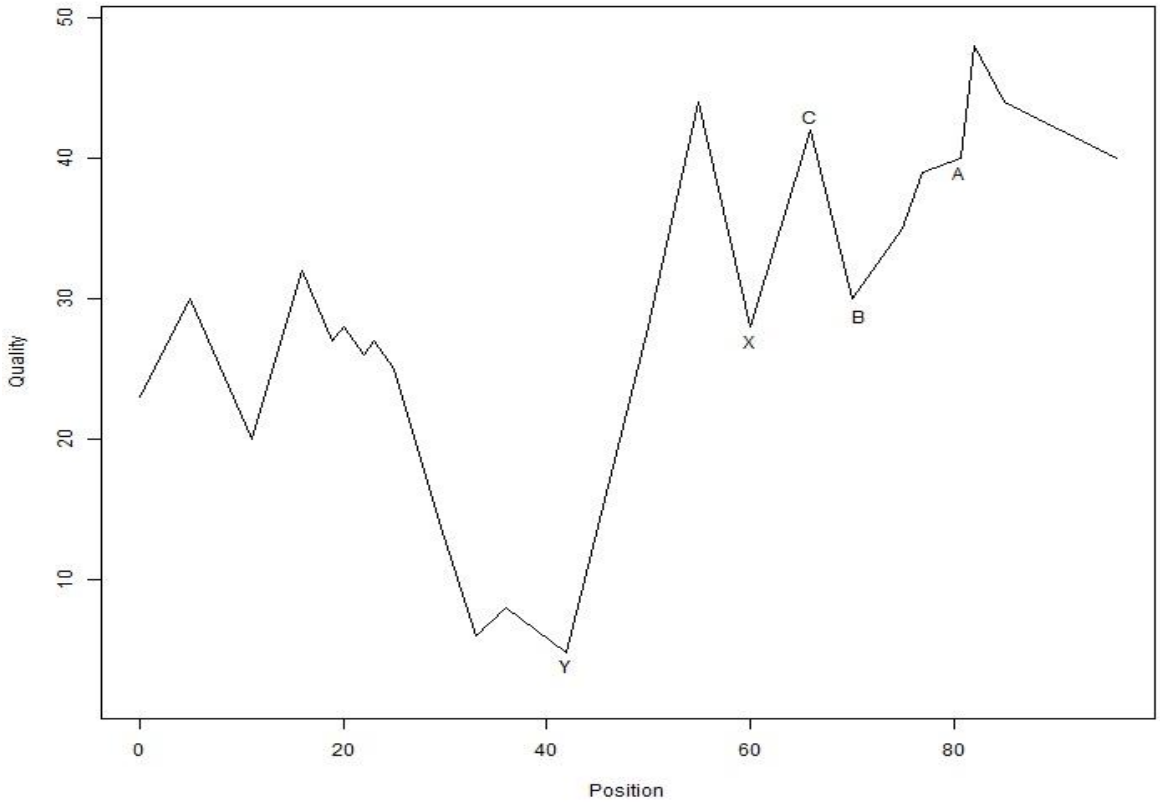

Figure 2.10 -- The shallow pitfall problem.

Here the vertical axis represents solution quality and the horizontal axis represents position in the space of potential (stable) solutions. The problem of finding an optimal solution is like the problem of getting a marble or metal ball into the deepest trough of a jagged surface. The ball easily rolls down from A to B but to get it to X and then Y requires that the whole 'mountain range' is shaken by an 'earthquake'.

If the metal ball starts at point A, it will rapidly roll down to point B and stay there; but point B is not by any means the deepest trough. If we shake it gently (adding some random disturbance) it will bounce around: sometimes it will go upwards, but the tendency will be for it to jump over barriers like point C and come to rest at a lower minimum like point X. However the lowest point is at Y. To make it likely that the ball reaches that global minimum, we would have to shake it fairly violently for quite a long time.

Scott Kirkpatrick of IBM (Kirkpatrick et al., 1983) have shown that the behaviour of a Boltzmann Machine in seeking an optimum set of weightings is similar to that of a ball rolling about on a bumpy surface. He introduced the concept of thermal disturbance for shaking such systems out of local minima into deeper ones, and showed that starting at a high temperature and gradually reducing it (simulated annealing) was effective. The fact that Boltzmann first studied the physics of such random thermal motion explains why his name was chosen for this computing procedure.

The Boltzmann Machine has brought neural modelling back to the forefront of AI. Such systems can be taught virtually any stimulus-response behaviour; but there is a price to pay for this flexibility. They are very slow. As Hinton himself says (1985):

> "Our current simulations are slow for three reasons: it is inefficient to simulate parallel networks with serial machines, it takes many decisions by each unit before a big network approaches equilibrium, and it takes an inordinate number of I/O pairs before a network can figure out what to represent with its internal units. Better hardware might solve the first problem, but more theoretical progress is needed on the other two. Only then will we be able to apply this kind of learning network to more realistic problems."

The Boltzmann Machine is extremely versatile, but it is not yet a practical device. It learns far too slowly for practical applications. For the present it is best viewed as an exciting idea thrown into the melting-pot of current proposals on how to design the next generation of parallel computers -- with suggestive implications for students of the nervous system.

2.4 SIGNATURE TABLES

The signature table (ST) approach to machine learning seeks to construct little boxes in feature space such that each box contains only (or mostly) one kind of pattern. It is a slightly more comprehensible process than parameter adjustment (PA), and is clearly akin to the widespread computing technique of indexing -- using attributes of an object to decide where to store that object.

2.4.1 Bledsoe and Browning's program

An early program of this type was the character recognizer of Bledsoe and Browning (1959).

In their system the image of a character was projected onto a 10 x 5 matrix and digitized. Each pixel in the matrix was given a binary value -- either on or off. The 150 pixels were paired up, at random, into 75 couples.

A pairing can be in one of only four states -- 00, 01, 10, 11 -- and these are used to address one of four different locations in memory. As each couple is connected to four memory words and there are 75 couples, this uses 300 words of storage. The words in this case are 36 bits long, with one bit allocated to each of the 26 capital letters and 10 digit characters. Initially all 300 words are filled with zeroes.

What happens when a character is presented to the system during training is that each couple responds in one of its four ways. For each couple one (and only one) of its four associated words is selected, and in that word the bit belonging to the character on display is set to one, as shown in Fig. 2.11.

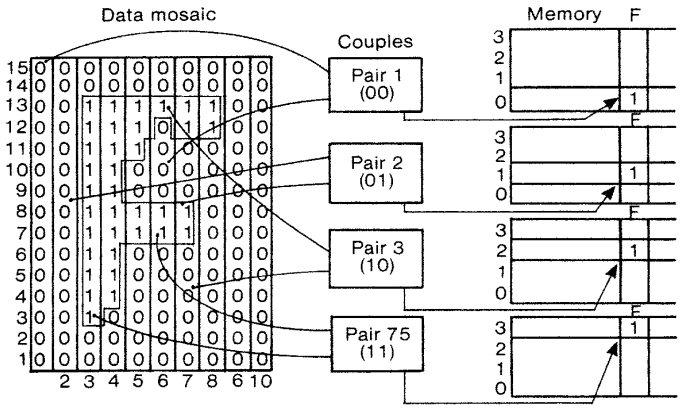

Figure 2.11 -- Bledsoe and Browning's program.

The diagram illustrates what happens when an F is presented (pattern 16) during the training phase. Each pairing responds in one of four ways, and thus selects one out of the four memory words to which it is connected. At that word, in the column (or bit position) reserved for F, a one is set. If F is the 16th pattern, then the bit is set at the 16th position, or column, within that word. Bits that never get set mark 'forbidden' subpatterns -- where a pairing has never been in the given state with the given pattern.

If F is the 16th pattern, and an F is presented, the 16th bit in 75 of the 300 words will be set -- one associated with each of the 75 couples. Which one, of the four, is determined by the state of that couple.

During the learning phase several examples of each possible pattern are presented, and the memory gradually fills with ones. However some bits are never set, because some couples cannot be in certain states given certain characters. For instance, it is hard to imagine an O or Q causing the central pair at row 8 column 5 and row 8 column 6 both to be on together. In effect the system learns about forbidden sub-patterns in the various characters, as defined by the state of pairs of pixels.

After training, when test characters are displayed, each of the 75 words addressed contributes one or zero to 36 totals, one for each possible pattern. The bit position with the highest total is chosen as the correct decision. Ties are resolved randomly.

A bit set in any column says that such a configuration of the relevant couple has been seen before for the character assigned to that column. If the bit is zero, it means that the configuration has never occurred with that character.

This is not a particularly sophisticated algorithm by today's standards, but it is instructive as an early example of some principles that are still in use -- in particular, the idea of using computed features of the input as addresses. Each pair of pixels functions as a simple feature detector, and the state of that pair is used to point to a position in memory associated with that state.

The system could be improved by storing integers, not just single bits, at each location, reflecting how often the feature-state had been encountered for each character, not merely whether or not it had been seen. But this would use up 36 x 300 = 10800 words in memory. As we shall see, ST systems often prove rather greedy in their storage requirements.

2.4.2 Samuel's checker-playing program

Arthur Samuel (1959, 1967) conducted two of the classic AI studies of machine learning. He used the game of checkers (or draughts) as a test-bed for ideas on searching, planning and -- more to the point -- machine learning.

His first program (1959) was essentially a PA system. It introduced some important ideas about game-playing by computer, but it was not so interesting as a learning machine -- although he did emphasize the role of 'selective forgetting' in rote-memory systems. In his 1967 paper, however, he introduced the key notion of 'signature tables' to describe a method he found more effective than adjusting the parameters of a polynomial evaluation function. This is the aspect of his work we concentrate on here.

At first glance, game-playing would seem an ideal arena for machine learning. It is, after all, quite easy to arrange for two programs, or two versions of one program, to play each other overnight and record the results. One might hope to leave the machine on all night and come back to find it playing at expert level.

The reason it is not this easy is that the credit-assignment problem is especially acute. At the end of a lost game, the program has to decide which moves were the bad ones. Various solutions have been proposed -- one of which is to retrace all the moves from the winner's point of view and regard all those moves where the loser would have played differently as mistakes. But in fact the winner is likely to have made mistakes too. If the winner makes six blunders and the loser makes seven, the loser is likely to pick up more bad habits than good ones. In general, there is no credit/blame assignment algorithm that is both simple and effective.

Samuel neatly sidestepped this dilemma by turning the game into a pattern recognition task. The problem then becomes one of looking at a board state (the pattern) and deciding which move is best (classifying moves). To provide grist for the learning mill he had several hundred thousand moves from published master-level games encoded on magnetic tape. Only moves by players that led to wins or draws were stored. Thus all moves by losers were ignored. The game-state together with the move actually chosen constituted a training instance. The program examined up to 180000 of these instances to form its signature tables.

There were actually three levels of the table in his experiments, as shown in Fig. 2.12.

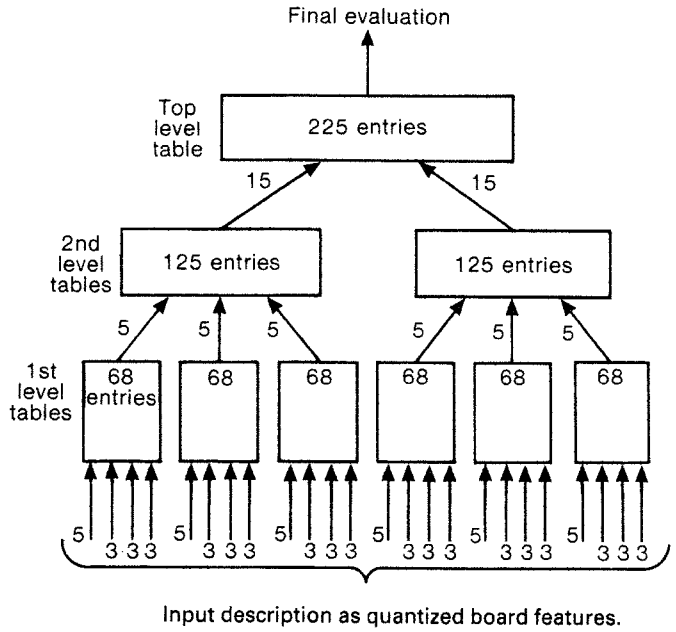

Figure 2.12 -- Signature-table hierarchy.

At the bottom level are 24 features which are computed from the board state. These are numeric measures of such factors as centre-control, piece advantage, threat to opponent's kings and so forth. Such features are used routinely in game-playing programs to describe the board state.

The learning process begins by quantizing these features into one of three or five levels. In effect the raw value is converted to one of low, medium and high (three levels) or one of very low, low, medium, high and very high (five levels). This enables it to be represented by a small integer.

The 24 quantized features are then divided into six groups of four, as shown in Fig. 2.12, one per table; and the state of each of the four features is used as an index into the table it belongs to. Thus the tables are 5 x 3 x 3 x 3 arrays containing 135 entries (but because checkers is a zero-sum game and the features are defined symmetrically, only 68 entries need to stored).

Each cell in the array is associated with a particular combination of feature values, and thus with a particular board configuration. This constitutes a kind of signature of that board state.

Given a game-state, each of the 24 features is computed and used to select a single cell in each of the six low-level signature tables. What actually happens during the learning phase is that the game-states corresponding to all the possible next moves are generated, and each of them produces a set of six signatures for the six tables. In fact each cell contains two numbers (initialized to zero) called D and A. When a signature is formed from a move that the expert did not pick, D is incremented by one. When a signature is generated by a move the expert did make, A is incremented by N, where N is the number of alternative moves that the expert could have made (but rejected) at that point. This gives greater positive weight to the moves the expert did choose than negative weight to those not played by the expert -- some of which may be quite good moves.

As and Ds are accumulated separately for all signatures in all tables. After a large number of examples, the program can compute an evaluation for any proposed board state (and hence for a proposed move) from the formula

$$C = (A - D) / (A + D)$$

where C is interpreted as a coefficient expressing correlation with the expert's choice. C can vary from -1 to +1, with higher values indicating better moves.

In this manner each signature (corresponding to a board configuration) points to a location in its table where the information can be found for estimating its value, by reference to numerous examples of expert play. But how are the six tables to be combined into an overall assessment? At one stage Samuel simply added their outputs up and took the total (in the same way as Bledsoe and Browning had done), but later he hit on the idea of applying the same procedure to them. This is the purpose of the second-level tables in Fig. 2.12. Just as the first-level tables store information about combinations of feature values, so the second-level tables hold information about combinations of signature-table values. Once the first-level tables have been filled, the process can be repeated, quantizing the output of the first-level tables (to five discrete levels in this case) and using them to index into two 5 x 5 x 5 tables where the values of various first-level combinations can be accumulated. In effect the second-level table learns the weightings to apply to its first-level tables.

The third-level table is filled in the same way. The outputs from the two second-level tables are quantized to 15 levels each, and used to address cells in a 15 x 15 array. Again, it is effectively weighting the contributions of the mid-level tables.

Samuel found the ST procedure far more effective than a comparable PA method using linear functions. After 184000 stored moves had been analyzed the correlation coefficient of the computer's evaluation with the expert's evaluation was nearly twice as high (0.48 to 0.26) with the signature tables than with parameter adjustment. In another test, on 895 unseen examples, the ST program rated the master's actual move as its first or second choice on 64% of the occasions. (By searching forward using alpha-beta minimaxing the program could do even better in actual play.)

The ST procedure gives better performance than a linear PA method because it is capable of handling non-linear interactions among the board features. These are actually quite common in practice.

Its chief defect is that the groupings are fixed. The selection of which four features should be grouped together to address a single table was done by Samuel. The program has no way of altering this choice, and an exhaustive search of all possible 4-from-24 groupings would be out of the question. If low-level tables were allowed to have three or five or six inputs as well, the number of combinations would become astronomical.

2.4.3 Michie's Boxes

Another system in the ST tradition but with a markedly different application was Boxes, designed by Donald Michie and colleagues at the University of Edinburgh (Michie & Chambers, 1968). This was a program that learned to balance a pole on a moving cart.

A small cart driven by an electric motor runs along rails carrying a pole hinged at the bottom. The cart must move back and forth to keep the pole from falling, rather like a Scottish Highlander preparing to toss a caber. In addition, the cart must not run off the ends of the track. This sort of set-up, illustrated in Fig. 2.13, is often posed as a problem for students of control engineering to solve using complex analogue circuitry.

[....]

Figure 2.13 -- The Boxes balancing act.

The Boxes computer system learned to control the movement of a wheeled cart on rails, so that it could keep the pole upright and avoid falling over the ends of the track for up to 30

minutes at a stretch. The computer's output could be either L or R, to send the cart left or right. Its inputs were the position and velocity of the cart and the angle and rate of change of the pole.

The program has two outputs, L and R, which cause the cart to move left or right, and four inputs. The input signals are shown in Table 2.4.

Table 2.4

Parameter	Range	Unit
Position of cart	-35 ... +35	inches
Velocity of cart	-30 ... +30	in/sec
Angle of pole	-12 ... +12	degrees
Rate of change of pole's angle	-24 ... +24	degrees/sec

These features are quantized, two with three levels and two with five. The intention is the same as with Samuel's program described in the previous section, to reduce an unrestricted numeric range to a range of small integers. This divides up the 4-dimensional space of possible states into 5 x 5 x 3 x 3 = 225 compartments. A typical situation might be: cart near right-hand end; cart moving leftwards; pole slightly inclined to the left; pole swinging towards the left. Each situation thus addresses its own box within the state-space.

One way of looking at this knowledge representation is to say the system starts with 225 rules in the

situation => action

format with the left-hand sides predefined but the right-hand sides (actions) to be determined by experience.

At the start of the learning process move-left and move-right decisions are distributed at random among the 225 compartments. As the program goes along it updates the decision frequencies in each box after each run, according to the length of time the pole stayed upright and the number of times the box was addressed during the run. In consequence, its relative preference between the two actions in each box varies. Eventually it learns to perform as an expert pole-balancer. After about 60 hours of practice, for instance, the system can balance the pole for 25 minutes at a stretch.

It is noteworthy that some regions in the state-space require counter-intuitive responses. Sometimes, when the cart wanders almost off the rails, it is necessary to push it a little further towards the danger zone in order to swing the pole in the opposite direction and then follow the pole to safety. Boxes was capable of learning this counter-intuitive skill.

Although this is a task that appears to demand little intelligence, it is very difficult to program a computer to perform equally well from first principles -- i.e. by using the laws of dynamics. The conventional solution using differential equations requires inordinate processing power to run in real time.

2.4.4 Aleksander's WISARD

Our final example of a signature-table mechanism is not a computer program at all, but a piece of special-purpose hardware. WISARD (Wilkie, Stonham and Aleksander's Recognition Device) was developed at Brunel University, and can be taught to distinguish, for example, smiling from frowning faces (Aleksander & Burnett, 1984). It is the basis for a commercially successful vision system that can detect sub-standard produce on a moving conveyor belt. The system operates in real time (25 frames a second) connected to a closed-circuit TV camera.

Briefly, WISARD works by assigning groups of eight pixels at a time (octuples) to selected banks of RAM (Random Access Memory). An octuple is essentially an enhancement of Bledsoe and Browning's couple: it acts as a feature detector which can be in one of 256 states (0 to 255) depending on the status of the pixels it monitors. During training, a one is stored at the location within its RAM-bank specified by the state of the octuple when a given image is present. Later, in the recognition phase, the presence of a one at the addressed location within that octuple's RAM-bank is evidence that the taught image is again present, since the same configuration was seen previously.

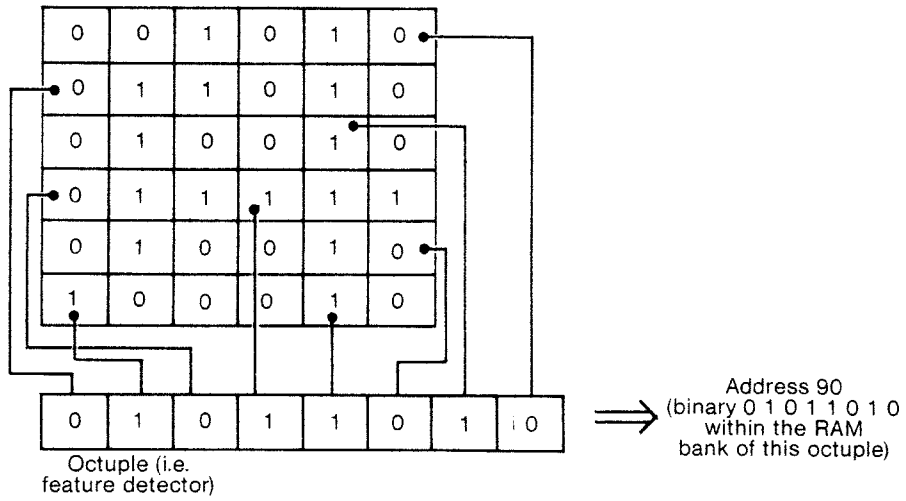

Fig. 2.14 -- WISARD schematic.
Each octuple is connected to 8 pixels and uses their status as a kind of address to select one out of the 256 RAM locations it controls. There may be thousands of octuples, or feature detectors. Quiz: is this an A or an H on display? Perhaps the computer can tell?

By using a very large number of octuples, the system becomes relatively impervious to spurious data in the image. Aleksander's system uses a 512-by-512 grid for the digitized image and more than 32000 octuples as feature extractors. Each octuple has its own bank of 256 memory addresses, so this requires over 8 million bits, or a megabyte, or RAM. Only in recent years have such memories become affordable.

The great advantage of WISARD is that all the RAM-banks are addressed in parallel. Thus it can work at high speed with moving TV pictures. By contrast, many 'state-of-the-art' vision systems running on conventional computers can take several minutes to process one image!

Note that the assignment of pixels to octuples is random, though it must be repeatable. The features are not pre-programmed.

Each octuple, or discriminator, is arbitrarily connected to 8 cells, and samples the status of these 8 pixels. The status is converted into a binary number such as 10110001 (177 decimal) and used as the address of one of the 256 memory locations within the bank to which that octuple is linked. During training a one is written at that location when the image is present. During recognition a one is read from that location when the octuple is in the same state.

Each octuple is evidence that the taught pattern is present when a one is read out and evidence of its absence when a zero is read out. The system decides by adding up the number of ones read out for each category and picking the one with the highest total. Aleksander has experimented with

providing positive feedback for the system, but essentially the decision algorithm is a voting procedure.

WISARD is interesting because it demonstrates that practical adaptive recognition systems can be built on relatively simple principles and because it shows that there is more to machine learning than programming computers. It works as a parallel computer, using microelectronic components designed for the computer industry in a novel way. As a program on a serial computer it would be horribly slow; as a parallel machine it is impressively quick.

2.5 SUMMARY AND CONCLUSIONS

We have reviewed a variety of black-box learning systems. What lessons have we learned?

2.5.1 The wheel of fashion

One of the most interesting points to emerge is the cycle of fashion.

Table 2.5

	Parameter adjustment	**Signature tables**
Old:	Perceptron	Bledsoe and Browning's program
New:	Boltzmann Machine	WISARD

Table 2.5 above shows the oldest and the latest example in each of our two categories. In hardware terms, they are worlds apart; but as far as the software is concerned, they are strikingly similar. Once someone has a good idea, it keeps re-surfacing in a multitude of guises.

Let us first compare the Perceptron with the Boltzmann Machine. It is clear that the Boltzmann Machine is a far more powerful design, since it has several layers of processing units with feedback loops and so on. Fundamentally, however, both systems share a common heritage, based on similar ideas about how to simulate the nervous system using threshold units.

It is interesting to note, moreover, that the Boltzmann Machine is not much more useful than the Mark I Perceptron. Even if it were realized in custom-built hardware it would not, in its present form, be an efficient learning machine.

One problem with the Boltzmann Machine is that it is not whole-hearted enough as a neural model. For example, the assumption that all interconnections are of equal strength in both directions is not grounded on any neurophysiological findings, but is convenient when it comes to proving theorems about the system's behaviour. It is reminiscent of the story about a drunk who, when asked why he is looking for his wallet under a lamp-post after dropping it on the other side of the street, replies: "because the light is better here".

The Boltzmann Machine will doubtless fall from grace, like the Perceptron, as its practical limitations become widely appreciated. But the idea of simulating the brain will not go away. People will continue to build neurologically inspired models, and one day may well construct devices of comparable power to the human brain -- probably by imitating it more slavishly than anything we have seen so far. But even then it is unlikely that they will truly understand how it works in a theoretical sense.

The lesson here is that the brain is a seductive, but unrewarding, model -- given our present level of understanding.

The contrast between WISARD and its predecessor, on the other hand, shows that much progress has been made. Why should WISARD be a practical success while connectionist devices like the Boltzmann Machine remain a pipedream? This is not an easy question to answer. It may be simply that it was designed with a practical application in mind. Or perhaps the secret is that WISARD matches up-to-date equipment (where we have made great strides) with a well-tried conceptual scheme (where we have not).

In any case it seems that some ideas that are rather antiquated -- in computing terms -- gain a new lease of life when implemented on modern hardware. Thus the designer of learning systems can expect to find useful ideas in systems that are several decades old.

2.5.2 The critics

The critic is the part of the learning system that has to evaluate its performance (as explained in Chapter 1). All the systems so far described try to simplify the critic's job as far as possible. The easiest thing is simply to compare the machine's answer with the teacher's, and several systems do just that.

Among the systems where the critic has a bit more work to do are Samuel's and Michie's. Both go to considerable lengths to turn multi-step tasks into single-shot tasks -- i.e. to remove the time dimension. This simplifies the credit-assignment problem, and in the present state of the art is probably a good plan to follow.

Michie's Boxes program maintains usage and length counts for the alternative actions (L or R) in each compartment of the state space. It probably has the most complex of the credit-assignment procedures used, but even this is relatively straightforward.

The lesson for system designers is: keep you evaluation measure as simple as you can.

2.5.3 Feature extraction

All the systems we have considered reduce information from a mass of raw data to a smaller number of computed features. It is far from clear, in general, how to reduce the amount of incoming data to a manageable level without throwing away vital information. Yet several of the systems surveyed (e.g. Pandemonium and WISARD) generate their feature detectors randomly, and still perform acceptably well.

This suggests that even if you do not know what features are significant, you can achieve good performance by letting the machine make them up. Indeed this technique has a distinct advantage: the machine is not biased to respond to only a few expected types of pattern.

2.5.4 The description language

All the systems in this chapter use numeric features to characterize input patterns. Thus an input is simply a vector of numbers. Some go as far as reducing input to an array of binary numbers (0 or 1).

This makes it possible to think in terms of a feature space in which examples are located; or (after the features have been reduced to a small number of discrete values) a state space. However it may take a good deal of pre-processing to turn raw data -- a speech spectrogram, say -- into a list of features. Although the feature vector is helpful for certain learning algorithms, it is not very convenient for people. In addition, as we shall see in the next chapter, feature vectors are extremely cumbersome when it comes to representing relationships between components of a pattern.

As for the rule description language, it is either a set of weighting factors (PA) or a table of some kind (ST). However effective these representations have proved, they are bound to be inscrutable. This is a severe liability as far as expert systems are concerned, since an expert system may have to explain its own reasoning. If all it can do is say "I looked it up in the table" or print out a row of coefficients (to six places of decimals!) then the user is likely to remain baffled.

This criterion alone means that black-box methods are the last resort when it comes to applying machine learning to expert systems, though it is usually possible to present table-lookup in a more comprehensible way than function evaluation.

2.5.5 Noise immunity

One very useful attribute that all the systems here share is a degree of tolerance for 'noisy' data. (Indeed, the Boltzmann Machine actually works better when the data is slightly 'noisy'.)

Noise is a concept borrowed from communications theory. Communications engineers frequently need to send messages along channels (such as short-wave radio) with background noise or interference. The signal is inevitably distorted during transmission. The receiver's problem is to pick out the signal from the noise.

By analogy, a pattern recognizer has to pick a category (the message) given input which may be spurious or garbled in some way. 'Noise' in this context covers a multitude of sins, for example:

-- the teacher sometimes gets it wrong;
-- instrument readings are not wholly reliable;
-- random processes affect the input signals;
-- the causal link between symptoms and diagnosis is tenuous at best.

Despite being such a rag-bag concept, it has proved useful in pattern recognition because the relationship between a pattern and its classification is seldom clear-cut. See Fig. 2.15.

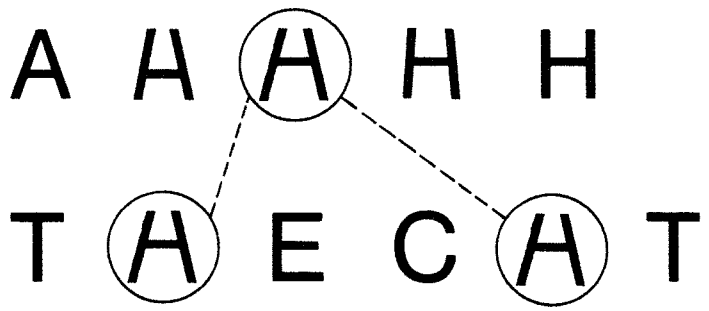

Figure 2.15 -- AH!

Pattern recognition systems have to deal with uncertainty in various forms. It is not really possible to say that there are 3 As and 2 Hs in the top line, or that there are 2 As and 3Hs. As the lower line (THE CAT) shows, the central letter can be pressed into service as A or H. This is one reason why the information-theoretic concept of 'noise' has been borrowed from the communications engineers.

None of the black-box systems expect 'perfect' training data. This makes them robust enough to cope with the fuzziness of the real world; though obviously there has to be a message hidden amidst the noise, or the best system in the world will find only phantoms and illusions -- as we do when gazing into a fire and seeing dragons, or hearing voices in a sea shell.

2.6 REFERENCES

Aleksander, Igor & Burnett, Piers (1984) *Reinventing Man*: Penguin Books, Middlesex.

Batchelor, Bruce G. (1974) *Practical Approaches to Pattern Classification*: Plenum Books, London.

Bledsoe, W.W. & Browning, I. (1959) Pattern Recognition & Reading by Machine: *Proc. Eastern Joint Computer Conf*. [Reprinted in Uhr (1966).]

Forsyth, Richard & Naylor, Chris (1985) *The Hitch-Hiker's Guide to Artificial Intelligence*: Methuen/Chapman & Hall, London.

Hinton, Geoff, E. (1985) Learning in Parallel Networks: *Byte Magazine*, 10, 4.

Kirkpatrick, S., Gellatt, C.D. & Vecchi, M.D. (1983) Optimization by Simulated Annealing: *Science*, 220.

Michie, Donald & Chambers, Roger (1969) Boxes: an Experiment in Adaptive Control. In Dale & Michie (eds.) *Machine Intelligence 2*: Edinburgh University Press.

Minsky, Marvin & Papert, Seymour (1969) *Perceptrons*: MIT Press, Massachusetts.

Rosenblatt, Frank (1958) The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain: *Psychological Review*, 65.

Rosenblatt, Frank (1962) *Principles of Neurodynamics*: Spartan Books, New York.

Samuel, Arthur (1959) Some Studies in Machine Learning Using the Game of Checkers: *IBM Journal of Research & Development*, 3.

Samuel, Arthur (1967) Some Studies in Machine Learning Using the Game of checkers, Part II: *IBM Journal of Research & Development*, 11.

Selfridge, Oliver & Neisser, Ulric (1959) Pandemonium: a Paradigm for Learning. In *NPL Symposium on Mechanization of Thought processes*: HMSO, London. [Reprinted in Uhr, 1966.]

Selfridge, Oliver & Neisser, Ulric (1960) Pattern Recognition by Machine: *Scientific American*, 203.

Sklansky, Jack & Wassel, Gustav (1981) *Pattern Classifiers and Trainable Machines*: Springer-Verlag New York and Berlin.

Uhr, Leonard (1963) Pattern Recognition Computers as Models for Form Perceptors*: Psychological Bulletin*, 60.

Uhr, Leonard (1966) ed. *Pattern Recognition*: Wiley, New York.

Uhr, Leonard & Vossler, C. (1961) A Pattern Recognition Program that Generates, Evaluates and Adjusts its own Operators: *Proceedings, Western Joint Computer Conference*.

3

Learning structural descriptions

The procedures of the last chapter have proved useful in a number of pattern-recognition tasks, but the knowledge they acquire tend to be rather opaque. We now turn from black-box learning techniques, where only the effectiveness of the system matters, to methods where the resultant knowledge is intended to be accessible to people as well as machines. This makes them more suitable for generating rules that can later be used as part of an expert system's knowledge base, because the knowledge in an expert system should be intelligible to humans.

In this chapter, therefore, we consider some description languages which are more sophisticated than those of the previous chapter. We will discuss representation schemes capable of expressing structural descriptions, where the relations between parts of an object are important as well as the elementary attributes (or features) of the object.

For example, in teaching a system to acquire the concept of 'arch' (Winston, 1975), it is necessary to capture the relationships of 'on-top-of' and 'touching', as illustrated in Fig. 3.1.

[....]
Figure 3.1 -- An arch and a near-miss.

> The most important property distinguishing the arch from a non-arch is the fact that the two pillars supporting the cross-piece are touching in the latter case. To express such properties succinctly requires a language in which two-term relationships (not merely single-term features) can be stated.

The things that distinguish an arch from a non-arch are the relationships among its components -- e.g. A-above-B, B-touching-C, C-below-A etc. To express such relations in terms of feature vectors would be hopelessly cumbersome.

3.1 THE DESCRIPTION LANGUAGE

As we have pointed out, the choice of representation for encoding a system's knowledge is at least as important as the details of the learning algorithm it uses. One of the most successful of recent discovery programs, Eurisko (Lenat, 1982), owes its success largely to its highly flexible description language. All Eurisko's concepts and heuristics are expressed in a common formalism, as 'Units'. Units are record-like structures that are modified by the discovery rules (other units). Simple syntactic changes in a unit usually lead to meaningful, and possibly valuable, new units. By contrast, small alterations in a conventional program or its data structures are likely to produce nonsense.

So before building a learning system it is essential to ensure that the description language is capable of expressing the kinds of distinction which will be needed. This is not a trivial problem.

It is very convenient if the representation for the input data is the same as that for the descriptions (or rules) but this is not always the case. Some representations that have been used in practice are given in Table 3.1.

A feature vector, as described in previous chapters, is just an array of numbers. Each number characterizes the state of one attribute of the input. In the simplest case the numbers are binary (0 or 1 only), meaning that the input is a bit-string.

Table 3.1

System	Input format	Rule format
Perceptron	Feature vector	Weight vector
Winston's program	Semantic net	Semantic net
ID3	Feature vector	Decision tree
AQ11/Induce	Predicate calculus	Predicate calculus
LS-1	Feature vector	Rule strings
BEAGLE	Data record	Boolean expression
EURISKO	'Unit'	'Units' (frames)

Winston's system learns simple concepts describing structures built from children's blocks, like the arch in Fig. 3.1. Both the training examples and the concepts are expressed as semantic networks. We discuss Winston's method briefly in section 3.5.

The ID3 induction program (Quinlan, 1979) uses feature vectors for the input but a tree structure for the decision rules that it builds up. (An example is given later in this chapter.)

The series of programs devised by Michalski and his associates including AQ11 and Induce (Larson & Michalski, 1978; Dietterich & Michalski, 1981) employ logical expressions in an extended predicate calculus notation to represent both input examples and class descriptions. The AQ11 program successfully induced descriptions from examples that enabled it to classify soyabean diseases better than an expert in agricultural biology. AQ11 and its successor, Induce 1.2, are described in section 3.4.

LS-1 is a program that we shall describe in the next chapter. It is a genetic learning algorithm (Smith, 1983). It was tested on a poker-betting task using simple feature vectors to represent the state of the game and fixed-length strings to represent production rules in a special language. These strings were manipulated by pseudo-genetic operators and represented the system's expertise by controlling its betting decisions.

BEAGLE is another evolutionary learning program described in the following chapter. It used 'flat-file' database records for its input examples and Boolean expressions, held internally as tree structures, for its rules.

Finally Eurisko as mentioned above, which is one of the most impressive current discovery programs, uses frame-like data structures called units to represent practically everything in the system -- including objects, concepts and the discovery rules themselves. Each field in a unit is called a 'slot' and describes one facet of the concept. For instance, the IS-A slot specifies subclass/superclass relationships. There is also a WORTH slot that specifies the value of a unit, on a scale from 0 to 1000. It is important to note that rules and meta-rules can be described as units as well as concepts: this is one of Eurisko's strengths. (To describe Eurisko's mode of operation would take too long: it is a most complex system. In a nutshell, however, what it does is wander around its conceptual space making small alterations to its concepts and rules and testing the consequences.)

From the variety of notations successfully used we can conclude that there is no ideal representation language for all machine learning problems. However, it is important that the representation used is expressive enough for the task in hand.

3.2 LEARNING AS SEARCHING

Assuming that we have an adequate description language, the next problem is to automate the generation of useful descriptions in that language.

One way of looking at this problem is as a search through the space of all possible descriptions for those which are valuable for the task in hand (Mitchell, 1982). The number of syntactically valid descriptions is astronomical; and the more expressive the description language, the more explosive is this combinatorial problem.

Clearly some way has to be found of guiding the search and thereby ignoring the vast majority of potential descriptions, or concepts, which are useless for the current purpose.

We can try to illustrate the link between the two fundamental AI notions of Search and Learning with a simplified weather-forecasting example. Let us suppose that we have a database of weather records and the job of the learning algorithm is to learn how to classify records on the basis of whether the next day will be fine or not. The machine must find a rule for making this discrimination.

Assume further for simplicity's sake that each record contains only four fields. These four variables are: rainfall in millimetres, sunshine in hours, maximum wind gust in knots, and pressure at noon in millibars. Thus two typical records from the training set (the instances used in forming the rule) might be as below.

(17-Apr-85)		(12-Apr-85)	
Rainfall	0	Rainfall	0.5
Sunshine	10.6	Sunshine	7.6
Windmax	15	Windmax	44
Pressure	1030	Pressure	1008
[Nextday	Fine]	[Nextday	Not-Fine]

The extra item, Nextday, indicates whether the next day turned out fine or not. For the present purpose, a fine day is defined as having

(Rainfall < 0.5) AND (Sunshine > 3.6)

i.e. less than half a millimetre of rain and more than 3.6 hours of sunshine. This is the value, derived from the following day's readings, that the system must learn to predict.

Rules can be composed by linking variable names and constants with the following operators.

Logical	AND, OR, NOT
Comparison	> = <

Thus the rule description language allows Boolean expressions such as

Sunshine < 4 AND Pressure < 1000

with brackets if necessary to avoid ambiguity. This is a relatively simple description language, but it suffices for our purposes.

Given this sort of training data and description language, the learning system can start with an initial description (which may be randomly created) and apply transformation operators to generate its successors. These successors are new descriptions for testing. The most important transformation operators are generalization and specialization. The process is illustrated in Fig. 3.2. For training data we use the 30 days of April 1985 (London readings). Of course in a real meteorological application, we might have tens of thousands of example cases measured on scores of variables.

Each node in the search tree is a decision rule that can be evaluated according to how well it distinguished days followed by rain from days not followed by rain. We use here an extremely simple evaluation score, the percentage of correct cases; but rules that apply to less than five cases (out of the 30 in the training set) are discarded. They are too rarely applicable. Our search strategy is to generate successors only from rules that have a high evaluation score i.e. to employ a best-first search method.

[....]

Figure 3.2 -- Learning as a search.

The top-level rule may be randomly generated or user-supplied. From then on, new rules are produced by applying generalization and specialization operators; such as

G1. Add OR condition	S1. Decrease LT-constant
G2. Increase LT-constant	S2. Add AND-condition
G3. Drop AND-condition	S3. Increase GT-constant
G4. Change AND to OR	S4. Change OR to AND
G5. Decrease GT-constant	S5. Drop OR-condition

Where do extra conditions like (Sunshine > 4) come from? One possibility is that there is a 'plausible comparison generator', guided by elementary statistical principles.

The diagram shows that this process can be viewed as a search through a network where the nodes are descriptions (rules) and the arcs are transformations that modify descriptions and thus generate new ones. At each step we show only four successors being generated -- two by generalization and two by specialization -- from the best rule, as defined by our simple evaluation function. In reality, far more successors would be created at each step.

In this context generalization of a rule means that it applies to more cases and specialization means that it applies to fewer. Thus the operator G1, which adds an OR-condition, makes the resulting expression cover a wider class of examples. On the other hand, the operator S3, which takes an expression of the form

Variable > Constant

and transforms it to

Variable > (Constant + k)

where k is some small increment, makes the rule more specific. It covers fewer cases. There are several methods of automatic generalization and specialization, as we shall see in section 3.4.

Note that the search defines a tree-structure, but the underlying search space is a network, because repeated applications of the transformation can re-generate ancestral rules. This corresponds to re-visiting a node in the network, and there is an example in the diagram, where

(Windmax < 25) OR (Sunshine > 4)

is generated at two different levels in the tree.

A number of searching methods have worked well with noise-free training instances. Dealing with 'noisy' data like the weather, however, is a more challenging problem. We shall confine our attention principally to systems that can be used with noisy data.

3.3 QUINLAN'S ID3

Quinlan's ID3 (Interactive Dichotomizer 3) is a natural step forward from the signature-table methods of Chapter 2 (Quinlan, 1982). Instead of creating a decision table, it creates a decision tree. It is not particularly robust in the face of noisy data, though it could be improved in this respect if it did not always seek a 'perfect' rule. The program works as follows.

(1) Select at random a subset of size W from the training set (the 'window').
(2) Apply the CLS algorithm (detailed below) to form a rule for the current window.
(3) Scan the whole database, not just the window, to find exceptions to the latest rule.
(4) If there are exceptions, insert some of them into the window (possibly replacing existing examples) and repeat from step 2; otherwise stop and display the rule.

This procedure actually throws away the latest rule and starts again from scratch on each cycle.

The method of window selection is sometimes termed 'exception-driven filtering'. The need for a window arises because the main database may contain hundreds of thousands of cases, and hence be too large to be processed (from backing store) in an acceptable time.

The CLS algorithm (Hunt et al., 1966) acts as a subroutine of the main program. CLS stands for Concept Learning System and derives originally from work done by experimental psychologists in the 1950s and 1960s. It first appeared as a proposed model of what people do when given simple concept formation tasks, and only later became a computer algorithm, due to Hunt and others. Thus it is an example of AI borrowing from psychology, rather than the other way round.

CLS works by first finding the variable (or test) which is most discriminatory, and partitioning the data with respect to that variable. Quinlan used an information-theoretic measure of entropy (i.e. surprise) for assessing the discriminatory power of each variable, but others have suggested different measures, e.g. the Chi-Squared statistic (Hart, 1985). Having divided the data into two subsets on the basis of the most discriminatory variable, each subset is partitioned in a similar way (unless it contains examples of only one class). The process repeats until all subsets contain data of only one kind. The end-product is a discrimination tree, which can be used later to classify samples never previously encountered.

ID3 trees performed well on King-Rook versus King-Knight chess endgame problems, where the data is clear-cut and free from uncertainty. Really noisy data, however, such as weather records, leads it to grow very bushy decision trees which fit the training set but do not carry over well to new examples. In the worst case it can end up with one decision node for every example in the training set!

ID3's main shortcomings are listed below:

(1) the rules are not probabilistic;
(2) several identical examples have no more effect than one;

(3) it cannot deal with contradictory examples (which are commonplace outside the rarified setting of chess endgames);
(4) the results are therefore over-sensitive to small alterations to the training database.

These objections would lose much of their force if ID3 stopped before it reached a subset with no counterexamples at all.

To give an illustration of ID3 in action, let us imagine that we are giving it data about the flags of various states in the USA. The objective is for it to learn how to distinguish states that joined the Confederacy in the US Civil War (1860-65) from those that stayed loyal to the Union. It is an artificial example, but it serves to illustrate the way the system works. In fact, there are reasons for thinking that confederate states might choose different emblems from those that remained in the Union.

First of all, here is the training data. We have picked 23 states that existed at the time of the Civil War, and augmented our set with the Union and Confederate flags themselves, as well as that of the District of Columbia (which is as federal as you can get, though not really a state at all). This gives 26 examples in all.

The data is presented in conventional feature-vector form, which is the way ID3 expects it. There are nine variables, plus the Type column (U or C) which is the one we want to predict. [Note added in 2018: variable Saltire was labelled Xcross in 1986 edition.]

Table 3.2

Flag	Stars	Bars	Stripes	Hues	Saltire	Icon	Humans	Word	Nums	Type
Union	50	0	13	3	0	N	0	0	0	U
Confederate	13	0	0	3	1	N	0	0	0	C
Alabama	0	0	0	2	1	N	0	0	0	C
Arkansas	29	0	0	3	0	N	0	1	0	C
Connecticut	0	0	0	5	0	Y	0	4	0	U
Delaware	0	0	0	6	0	Y	2	4	2	U
Florida	0	0	0	6	1	Y	1	15	0	C
Georgia	13	1	0	3	1	Y	0	3	1	C
Illinois	0	0	0	6	0	Y	0	6	2	U
Iowa	0	2	0	5	0	Y	0	10	0	U
Lousiana	0	0	0	4	0	Y	0	4	0	C
Maryland	0	12	0	4	0	N	0	0	0	U
Massachusetts	1	0	0	4	0	Y	1	6	0	U
Mississippi	13	0	3	3	1	N	0	0	0	C
New Hampshire	9	0	0	5	0	Y	0	7	1	U
New Jersey	0	0	0	5	0	Y	2	3	1	U
New York	0	0	0	6	0	Y	2	1	0	U
North Carolina	1	1	2	4	0	N	0	3	4	C
Ohio	17	0	5	3	0	N	0	0	0	U
Rhode Island	13	0	0	3	0	Y	0	1	0	U
South Carolina	0	0	0	2	0	Y	0	0	0	C
Tennesee	3	2	0	3	0	N	0	0	0	C
Texas	1	1	2	3	0	N	0	0	0	C
Virginia	0	0	0	5	0	Y	2	4	0	C
Wisconsin	0	0	0	5	0	Y	2	2	1	U
Washington DC	3	0	5	2	0	N	0	0	0	U

The variables used are as follows. Stars is a count of the number of stars on the flag. Bars are vertical lines. Stripes are horizontal lines. Hues is the number of colours in the flag. Saltire indicates the presence of an X-shaped cross on the flag. Icon is Y if there is a pictorial design and N if the flag is

purely abstract. Humans is a counter of the number of human figures depicted on the flag. (If Icon=N then obviously Humans=0.) Word is another counter, of the number of words appearing in the flag (as a motto or slogan). Finally Nums gives the number of numbers represented: some states put dates, like 1848, on their flag. (Example flags are shown on the back cover of this book.)
Note that this input description language does not allow us to express structural relationships, such as 'a white star over a blue star'. This is one of the problems with feature-vector notation.

CLS begins by looking for the most discriminatory variable, in order to create the root of the decision tree. To do this it constructs a number of frequency tables, such as those shown in Table 3.3.

To make these contingency tables the program must pick one or more thresholds when dealing with numeric variables. The thresholds 0 and 1 were tried with Stars, giving (Stars > 1) as one test and (Stars > 0) as another. The second one turned out to be slightly better, but neither was much use. In fact the best test turns out to be

(Saltire > 0)

which is only true for Confederate flags, though it is false for both kinds of flag. (The original ID3 could not cope with unrestricted numeric attributes, but it can easily be extended to do so, as we have done here.)

Table 3.3

Test	C	U
Stars > 1	5	5
Stars <= 1	7	9
Stars > 0	7	6
Stars <= 0	5	8
Stripes > 0	3	3
Stripes <= 0	11	9
Saltire > 0	5	0
Saltire <= 0	7	14

The equivalent test (Saltire >= 0.5) is then established as the root of the tree, as shown in Fig. 3.3.

For the next stage, the left branch can be left alone: it only contains one type of data (Confederate). But the right branch needs to be further subdivided, using essentially the same method on the subset of 21 cases for which the top-level test is false.

The next best discriminator, for that subset, turns out to be

(Hues < 4.5)

which tests how many colours the flag has on it. This divides the 21 remaining cases by picking off one Confederate state and eight Union states when the rue is false. The other subset, when the rule is true, breaks 6:6 and has to be further subdivided.

One of the weaknesses of ID3 appears at this point. It always subdivides the subsets until no single exception remains. This striving for 100% correct rules can cause ID3 to generate very bushy trees whose nodes contain very few examples. These are unlikely to be statistically reliable when the tree is later used for classification. Furthermore, large branchy tree structures are difficult for people to comprehend.

[Note added in 2018: To avoid extreme subdivision, the procedure used to create the diagram (Fig. 3.3) was used with a settings that limited the maximum depth of the tree to four, and avoided diving nodes with only a single exception, thus this tree isn't identical to the 1986 version.]

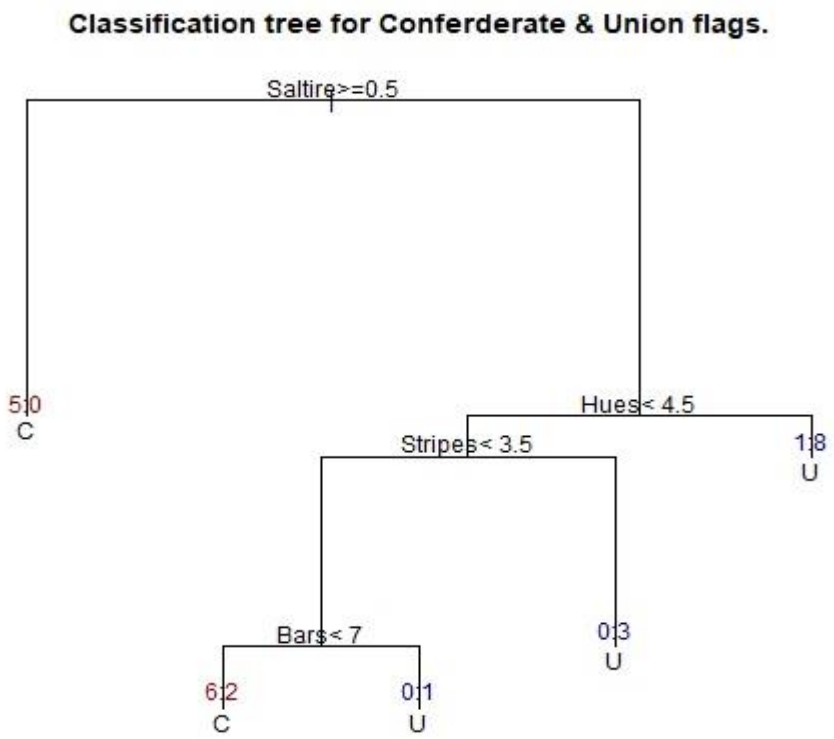

Figure 3.3 -- Example discrimination tree.

Here left branches apply when the test is true and right when it is false. The numbers falling into each of the five terminal (or leaf) nodes are shown just above the label (C or U) for that node, with C first. Red indicates a Confederate majority and blue a Union majority.
[Note added in 2018: Caption text, and next but 1 paragraph, amended slightly to decribe this tree which was created in R using the rpart() procedure.]

Once the tree has been grown, it can be used to make decision about new examples, by applying the tests and following down the appropriate branches to a terminal node. Thus if we asked, in effect, "is Britain a confederate state?" it would reply Yes, because the British flag does have a Saltire. (Ask a silly question . . .)

More relevantly, to decide whether the state of Washington (not D.C.) was Confederate, the system would ask: does it have a Saltire? It does not, so the number of Hues would be considered. It has six (counting black and white) so it would be filtered to the right branch and thus considered a Union state, because 8 of the nine examples falling into that leaf node are Union states. (It surely would have been, if it had been founded early enough, being about as far as you can get from the old Deep South.)

It will be seen that ID3 has much in common with the signature-table methods we encountered earlier. The decision tree can be thought of as a compressed signature table: the tree could be held

as a multi-dimensional matrix, except that it would take far too much room. The decision tree is also more natural for human readers. We are accustomed to using such trees in biology and other fields.

The problem of excessive subdivision could be cured simply by stopping early. For instance, any subgroup that contained fewer than, say, 5% of the training examples could be deemed too small for further division. When the tree was used to categorize new cases and such a node was reached, the system could give a probabilistic answer. Thus, in the diagram, on reaching the node with 8 union and 1 confederate instances, the system would respond with 0.89 Union instead of saying 'don't know' -- as assumed with the example of Washington state, above. Realistically, such a group is too small to split up any further. There are many domains where certainty is not attainable, and a probabilistic answer based on a reasonable sample is preferable to an exact answer based on a tiny sample.

A more serious problem concerns the nature of the description language itself. The decision tree is actually a rather restrictive language. All tests have to be in the form of a comparison between one variable and one constant, such as (Hues > 3). Tests like

(Stars >= Stripes)

and

(Hues > 4) OR (Humans > 0)

might prove to be useful, but the system could never find them. They are literally inexpressible.

This is the price paid for ID3's efficiency. It is relatively quick, but this speed is purchased at a cost. Clearly a more expressive description language would make the tree-growing far more complex; but the poverty of its description language places the burden of devising an effective set of descriptors squarely on the user. Any preliminary calculations or logical operations have to be incorporated in the attributes of the input data before running the program.

(ID3 forms the basis of a package marketed as 'Expert-Ease'.)

3.4 AQ11 AND INDUCE

The series of programs developed by Michalski and his colleagues at the University of Illinois use more powerful description languages than ID3.

3.4.1 AQ11 and the soybeans

The program AQ11 (Michalski & Larson, 1978; Michalski & Chilausky, 1980) is the one which found better rules for soybean disease diagnosis than a human expert. For that particular experiment, they collected 630 questionnaires describing diseased plants. Each plant was measured on 35 features (called 'descriptors') as in the abbreviated record below.

Environmental descriptors

Time of occurrence	= July
Plant stand	= normal
Precipitation	= above normal
Temperature	= normal
Occurrence of hail	= no
Number of years crop repeated	= 4
Damaged area	= whole fields
[22 descriptors omitted for brevity]	

Plant descriptors

Conditions of seed	= normal
Mold growth	= absent
Seed discoloration	= absent
Seed size	= normal
Seed shrivelling	= absent
Conditions of roots	= abnormal
Diagnosis	: Brown Spot

The 36th descriptor is the diagnosis of an expert in plant biology. There were 15 disease categories altogether.

AQ11 generated rules in a language call VL1, where a description is a set of terms called 'selectors'. The rule below (D3), which the system produced for classifying Rhizoctonia Root Rot, illustrates the language.

D3: [leaves = normal] [stem = abnormal]
[stem cankers = below soil line]
[canker lesion color = brown]
OR
[leaf malformation = absent] [stem = abnormal]
[stem cankers = below soil line]
[canker lesion color = brown]

This rule consists of two descriptions linked by an OR: it is a disjunction. Each description happens to consist of four selectors, such as [stem = abnormal], which are linked by logical ANDs. That is to say, AND is implied between selectors, so that a description in VL1 is a conjunction of terms. Each selector compares one variable with a constant (or range of constants).

As a matter of fact, this particular disjunction is trivial. The only difference between the two descriptions, [leaves = normal] versus [leaf malformation = absent], would be unnecessary if the system realized that one was a special case of the other.

This diagnostic rule (D3) and 14 others were generated from 290 training instances. The training set was selected from the 630 cases by a program called ESEL, which picks examples that are 'far apart', or different from each other, to give a broad coverage. Michalski did not report what AQ11 would do with a randomly chosen training set, so the effect of this selection strategy is unclear.

The 15 computer-generated rules were used to classify the remaining 340 cases, with great success. Whereas the human expert's rules gave the correct first choice disease on 71.8% of the test cases, AQ11's rules were gave correct first choice on 97.6% of the unseen cases.

AQ11 works in an incremental fashion, each step appending another conjunctive term (i.e. a new selector) starting off from a null description. The idea is to introduce new items of evidence one at a time, or a few at a time, and extend the growing rule to deal with them. The AQ11 method can be outlined in four major steps.

(1) Pick a new training event.
(2) Generate a 'star' of new terms by extending the current description to cover as many positive events and as few negative events as possible (ideally no negative ones at all).
(3) Retain the most preferred description (e.g. the simplest) according to pre-specified criteria.

(4) If all relevant events have been covered, keep the description; otherwise go back to step 1.

If a single conjunctive description cannot be found to cover all positive examples, AQ11 will generate several, linked by ORs, as we saw with rule D3.

The 'star' method extends the current hypothesis (i.e. description of a given category) to cover the latest event or events. First it isolates all facts inconsistent with the hypothesis, both positive and negative exceptions. Then it derives two new descriptions -- one that covers the positive cases that were not covered by the old hypothesis (D+) and another that covers the negative cases that were covered by the old hypothesis (D-). Next it updates the hypothesis by appending the D+ terms and removing the terms that caused it to include the D- events. Finally the descriptions so generated are tidied up.

In the soybean work, the system made a complete pass through the data for each disease type, treating cases of that disease as positive examples and all other cases as negative examples. It is also possible for AQ11 to treat previously generated rules as negative examples: this enables it to come up with non-overlapping rules where the categories are mutually exclusive.

3.4.2 Induction by Beam Search

AQ11 rules start off very general and become more and more specific. It adds new terms to exclude negative examples, while still covering as many positive cases as possible. The successor of AQ11, Induce 1.2 (Dietterich & Michalski, 1981), works the other way round. It starts with very specific descriptions and keeps on generalizing. It also uses a richer description language called VL2, a form of 'annotated predicate calculus'. This allows quantifiers, functions and relational predicates with more than one argument. A VL2 description of the arch in Fig. 3.1, for example, could be

(EXISTS A, B, C)
[Touching(A,B)] [Touching(A,C)] [Ontop(A,B)] [Ontop(A,C)]
[Shape(A) = Pyramid] [Shape(B) = Block] [Shape(c) = Block)] .

Induce explores the description space by the method of Beam Search. This is a modified best-first strategy that preserves a small number of descriptions at each stage.

The progress of a Beam Search is illustrated in Fig. 3.4. Here we show the search fanning out upwards. At each level only the seven best nodes are retained. All the other nodes are pruned away, and do not generate any successors. Each node in the diagram represents a description. Upper levels are more general (i.e. cover more cases) than lower ones. In this case the tree grows upwards only. As in Fig. 3.2, the same node/description can sometimes be reached by two or more routes.

[....]
Figure 3.4 -- Beam Search.

> In this Beam Search, only the best 7 nodes are preserved at each stage and allowed to generate successors. The generation process proceeds upwards towards more general descriptions. Nodes shown with more than one 'parent' are descriptions created in more than one way: this can easily happen.

Induce employs the Beam Search, as outlined below.

(1) Set H to contain a randomly chosen subset of size W of the training instances. (These are also rules which happen to be very specific.)
(2) Generalize each description in H as little as possible.

(3) Prune implausible descriptions, retaining the best W only. The best are those that are simple and cover many examples; the worst are those that are complex and cover few examples.
(4) If any description in H covers enough examples, print it out. If H is empty or enough rules have been printed, stop; otherwise continue from step 2.

We have been deliberately vague here about the criteria for evaluating descriptions, and hence deciding which ones to prune. This is the job of the Critic (see Chapter 1). Obviously rules should be brief and they should cover many positive and few negative examples, but various trade-offs are possible. Induce provides several preference measures which can be combined to give an evaluation function tailored for a particular application.

Induce works from specific descriptions to more general ones. It is easy to start it off with an initial set because it uses the 'single representation trick'. In other words, the VL2 language expresses both the rules and the training instances. A training instance can be regarded as a highly specific rule -- a class with only one member (or possibly a few members if there are repeated examples).

To turn an event description into a rule, or a rule into a more general one, Michalski's program uses a variety of generalization operators. We can illustrate some of them by encoding another flag (this time the national flag of Canada) in a version of VL2.

(EXISTS X, Y, Z)
[Background = White] [Hue (X) = Red] [Hue (Y) = Red] [Hue (Z) = Red]
[Type (X) = Bar] [Type (Y) = Maple Leaf] [Type (Z) = Bar]
[Left-of (X, Y)] [Left-of (Y, Z)] [Left-of (X, Z)]
[Width = 50] [Height = 25] [Hues = 2]

As well as encoding simple features, such as [Width = 50], VL2 can express structural relationships, such as [Left-of (Y, Z)], which is a step forward from the flag descriptions we used in the previous section. To put it another way, it can handle binary predicates as well as unary ones. It can in fact handle predicates with more than two arguments, if required.

Let us now consider some ways of generalizing this description.

(1) Dropping Conditions:
e.g. delete the last three lines of the above description to create a new description that applies to any flag with a white background and three red objects or regions.
(2) Internal Disjunction:
e.g. [Hue (X) = Red] => [Hue (X) = Red, Blue]. This now also covers a variant of the Canadian flag with a blue bar on the left-hand side.
(3) Relax a Condition:
e.g. [Height = 25] => [Height > 24]
or [Height = 25] => [Height < 26]
These effectively introduce disjunctions.
(4) Make a Constant into a (Don't-Care) Variable: e.g. [Type (X) = Bar] => [Type(?) = Bar], where the question mark stands for any object, meaning that any object can be a bar. Again, this is a way of introducing a disjunction.

Other methods of generalization were described in section 3.2.

The problem is not that machines cannot generalize. On the contrary, there are too many ways of generalizing. This is why Induce only makes minimal generalizations on each cycle (step 2). The successors of a node are produced by applying one generalization operator in only one way.

For example, the dropping-conditions method (no. 1) would cause the generation of 13 successors to the Canadian flag description, since it contains 13 terms. Each successor would differ from the original in having one condition dropped. The program would not drop two or more conditions at once. Nor would it drop a condition and apply another generalizing rule (such as relaxing a condition) in the same step. For that to happen would require at least two cycles. This is mainly a question of efficiency: if all generalization operators were applied in all possible ways, there would be an astronomical number of new nodes.

The trouble is that even single-change generalizations can lead to an enormous proliferation of descriptions at each stage, especially with an expressive language like VL2. It is highly likely that some which are unpromising in themselves, but which would lead to good descriptions a few more steps down the line, will be pruned away. This ensures that the Induce algorithm is non-optimal (though in practice it may perform very well).

One final point about Induce, not so far mentioned, is that it can run the search in two distinct phases. In the first phase it searches the 'structure-only space'. That is to say, it ignores the unary attributes and does a Beam Search using only multi-argument predicates, such as Ontop or Left-of. Then, having found a set of descriptions using relational predicates only, it enters a second phase, which is a search of the 'attribute-only space'. At this point the unary features are considered. It is not yet clear, however, whether this two-stage approach is generally useful.

3.5 OTHER SYSTEMS

Several other programs have been developed for learning concept descriptions. We do not have space to describe them here in detail. Nevertheless, some of them deserve at least a mention.

3.5.1 Meta-Dendral

The Meta-Dendral system (Buchanan, 1976; Buchanan & Mitchell, 1978) has discovered several rules of chemistry that were previously unknown. Specifically it found new cleavage rules that describe how organic molecules, of the ketoandrostane group and others, fragment in a mass-spectrometer. These rules were later incorporated into the knowledge base of Dendral (Buchanan & Feigenbaum, 1978), one of the classic expert systems. Dendral interprets mass-spectra. Fig. 3.5 shows a typical mass-spectrum.

These patterns are produced by an instrument that bombards chemical samples with accelerated electrons, causing them to break up. The fragments are then passed through an electromagnetic field that separates out the fragments with low charge and high mass, which are not deflected much by the field, from those of low mass and high charge, which are deflected considerably. This gives rise to the plot of intensity against mass-to-charge ratio.

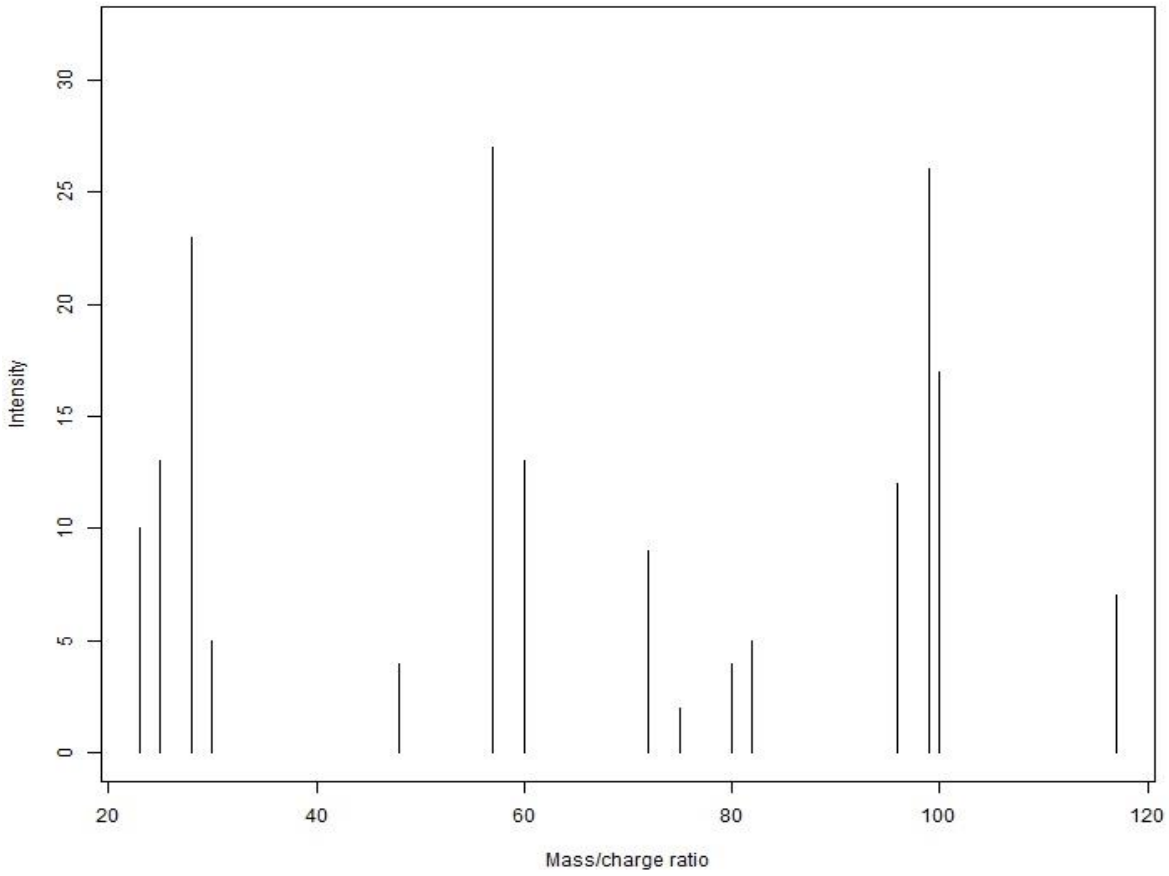

Figure 3.5 -- Mass spectrum.

Organic chemists use such mass spectrograms, together with a knowledge of the chemical composition of the compound, to deduce its chemical structure. Different structures have different characteristic peaks at different mass-to-charge levels.

A trained chemist can look at such a plot and identify the molecular structure of the compound that produced it by noting where the peaks are. So can Dendral. In doing so it tests its own hypotheses against a simulation of a mass-spectrometer. Meta-Dendral was designed to enhance this internal model of the device by discovering additional fragmentation rules.

Meta-Dendral contains two main programs, Rulegen and Rulemod. Rulegen performs a relatively crude search of the space of potential cleavage rules, using only positive training instances. It moves in the direction from general to specific. It starts with the most general rule possible (that some atomic bond will break) and gradually specializes it. At each stage it creates offspring of existing rules by making them more specific in various ways -- i.e. stating more precisely which bonds will break under what circumstances.

A descendant rule is retained if

-- it predicts fewer fragmentations per molecule than its parent (i.e. is more specific);
-- it still predicts fragmentations for at least half of the training molecules;
-- it predicts fragmentations for as many molecules as its parent (unless the parent was 'too general').

Rulemod takes the rules produced by Rulegen and performs minor alterations designed to improve their performance -- both by generalizing and specializing. In particular, it takes account of negative evidence.

Meta-Dendral can handle noise, which can arise from impurities in the samples, from imperfections in the instrument and from errors introduced by the program that transforms the training instances into a suitable form for processing. In addition, it has achieved respectable performance: some of its discoveries were written up and published in a chemical journal (see Michie & Johnston, 1985).

The real weakness of Meta-Dendral is the fact that its description language is specifically designed to be good for expressing rules about molecular structures and how they split up -- and for virtually nothing else.

3.5.2 Bringing home the Bacon

Another discovery system worthy of note is the Bacon series of programs (Langley, 1977 ,1981).

One of these programs, Bacon.4, 'rediscovered' -- among other things -- Ohm's Law, Archimedes' Principle of Displacement, Newton's Law of Gravitational Attraction and almost all of 19th century chemistry. (It has not so far discovered any 21st century chemistry.)

Bacon.4 is presented with training instances such as Table 3.4.

Table 3.4

Planet	Day	Year	Distance	Diameter	Mass	Moons
Mercury	58.00	0.24	0.39	0.38	0.05	0
Venus	244.00	0.62	0.72	0.95	0.82	0
Earth	1.00	1.00	1.00	1.00	1.00	1
Mars	1.03	1.88	1.52	0.53	0.11	2
Ceres	999.99	4.60	2.77	0.08	0.00	0
Jupiter	0.41	11.86	5.20	11.19	318.35	16
Saturn	0.43	29.46	9.54	9.41	95.30	15
Uranus	0.67	84.01	19.19	4.06	14.60	5
Neptune	0.75	164.80	30.07	3.88	17.30	2
Pluto	6.38	248.40	39.52	0.24	0.08	1
Trogstar Beta	2.22	680.00	77.22	16.48	444.44	4

Here the length of day, length of year, mean distance from the Sun, diameter, mass and number of satellites for all the members of the Solar System have been tabulated. (We have taken the liberty of including data for Trogstar Beta, the totally inconspicuous brown dwarf companion of our sun, which has not yet been discovered -- even by Bacon.4!)

To emulate Kepler, and discover that the square of the orbital period (Year) of a planet is proportional to the cube of its distance from the sun, the program has to note concomitant variations. (See Chapter 1 on J.S. Mill.) For example the length of day appears to vary inversely with the diameter, while the year and the distance vary together. This sort of observation leads Bacon to create new columns in the table, formed by fairly straightforward arithmetical combinations of existing ones using division, multiplication, etc.

Bacon proceeds step by step in its search for constancies. This is shown in Table 3.5, where all the variables apart from Y (year) and D (distance) have been ignored.

Table 3.5

	Y	D	Y/D	(Y/D)/D	((Y/D)/D)*Y	(((Y/D)/D)*Y)/D
Mercury	0.24	0.39	0.62	1.61	0.39	1.00
Venus	0.61	0.72	0.85	1.18	0.72	1.00
Earth	1.00	1.00	1.00	1.00	1.00	1.00
Mars	1.88	1.52	1.23	0.81	1.52	1.00
Ceres	4.60	2.77	1.66	0.60	2.76	1.00
Jupiter	11.86	5.20	2.28	0.44	5.20	1.00
Saturn	29.46	9.54	3.09	0.32	9.54	1.00
Uranus	84.01	19.19	4.38	0.23	19.17	1.00
Neptune	164.80	30.07	5.48	0.18	30.04	1.00
Pluto	248.40	39.52	6.29	0.16	39.51	1.00
Trogstar Beta	680.00	77.22	8.81	0.11	77.55	1.00

The figures are correct to two decimal digits.

Bacon's first step, having noted that Y and D vary together, would be to divide one by the other, forming the Y/D column. This still varies with D so the next column, (Y/D)/D, is created. Now it has overdone it, but it notices that the new column varies inversely with Y, so it multiplies by Y, giving the fifth column ((Y/D)/D)*Y. This now agrees almost perfectly with D. Eventually it finds a column like the last one, with a constant value. That is what it was looking for. It can be expressed more clearly as (Y*Y)/ (D*D*D). In a sense, the program has recapitulated Kepler's discovery that Y squared equals D cubed.

For exposition, our table has cut out the many blind alleys and red herrings which Bacon would probably explore. (Kepler, too, spent plenty of time on such things.) For example the program, given this data, might well go looking at the relationship of distance with rank position -- perhaps even to the extent of re-formulating Bode's notorious 'law', or something like it. Or it might try to relate the number of satellites to the mass of a planet and its distance from the sun. (Readers may care to investigate this for themselves.)

Bacon, like most scientists, has rather strong preconceptions about what form scientific laws should take -- i.e. that the relations between significant quantities should be mathematically simple. Human scientists often express this as an aesthetic principle: that Nature (or God) is clever but not devious; or that the beauty of Truth is its simplicity. Philosophers, however, sometimes take a different view, namely that the human mind is only capable of discovering certain rather simple regularities in nature.

The main drawback of Bacon.4 is that it cannot cope with noise. It needs a training set without significant exceptions, and thus cannot find approximate statistical laws.

[Note added in 2018: Bacon.4 in fact was able to cope with a degree of noise in the sense of having a tolerance parameter for deciding whether a column was constant.]

3.5.3 Winston's work

Another concept learning system (Winston, 1984) which unfortunately is not much good at coping with noisy data is Winston's program. Still, his work is interesting from our point of view because it has been influential in stressing the importance of 'near misses' (cases that almost but not quite

count as positive examples) and the importance of a good training sequence. In human terms, this amounts to saying that the role of the tutor is critical for effective learning.

One of the tasks Winston's program performed was learning to distinguish blocks-world objects, such as we saw in Fig. 3.1. The basic algorithm works as follows.

(1) Let the description of the first example (which must be a positive case) be the initial rule.
(2) For all subsequent instances:
If the instance is a near miss, specialize the rule to exclude the latest example;
If it is a positive instance not covered by the rule, generalize the rule to include the last example.

Nothing happens if the current rule covers the latest case. The algorithm only learns from its mistakes.

Notice that this is a strictly stepwise procedure. It maintains a single hypothesis and works with one training example at a time. Although it has trouble coping with inconsistencies in the training data, it is very efficient in that it does not need many passes over a large database (as many other systems do).

It is also interesting because it uses semantic networks to describe objects and the rules for classifying objects. This is an idiosyncratic choice of representation -- illustrating the fact that there is not yet a consensus about the right kind of description language in this field.

3.5.4 Rulestorming

Another interesting rule-generator is the system described by Quinqueton and Sallentin (1983). Their method consists of iterating an algorithm composed of three modules, optionally followed by a fourth phase, called 'rule-storming'. The three main modules are: expansion, selection and compression.

Their input language is as basic as it can be. Data are bit-strings, recording truth values for a number of logical features. One of these is treated as the dependent variable -- the one to be predicted.

The output rule is a logical combination of the variables using only symmetric operators, namely conjunction (AND) and logical equivalence (EQV). The method works as follows.

(1) Expansion: combine each pair of descriptors (logical features) with a logical operator in five permissible ways to create new, derived, descriptors. Thus from two basic features (A, B) the descriptors

A & B, A & ~B, ~A & B, ~A & ~B, A = B

would be created.
(2) Selection: eliminate the descriptors that correlate poorly (according to Chi-Squared or some similar measure) with the dependent variable.
(3) Compression: attempt to combine associated descriptors into clusters so that only one representative of each cluster needs to be preserved for the next stage (or just a few).

These steps are iterated. At the end of the process a rule-storming phase may take place. This attempts to construct a hierarchy of the surviving descriptors, and to devise a voting procedure for combining them in subsequent forecasting.

This method is relatively resistant to noise, and it does not make the assumption that the descriptors are metrical: they can be based on underlying nominal, ordinal or metrical values. Apparently it has been used to forecast earthquakes.

However, with a large number of descriptors the combinatorial problem in the expansion phase is quite severe. Moreover, the system uses an impoverished input description language. Considerable work is required to make raw data suitable for the rule-induction program. The rule language is not particularly comprehensible either.

3.6 CONCLUSIONS

Several themes emerge from our study of rule learning programs. Among the most important is the idea of

Conjecture + Refutation

which is said to be at the heart of all scientific discovery (Popper, 1959). In one way or another, all the systems described here generate potential solutions and then test them, typically discarding the majority.

There are various methods of generation, various measures of quality for the generated rules, various ways of handling training data, and so on. For example, there are model-driven (top-down) generators, guided by prior assumptions about the form of hypotheses; and there are data-driven (bottom-up) generators, guided by patterns in the training data. However, the idea of learning as search provides a framework within which these divergent approaches can be unified.

The diagram in Fig. 3.6 shows a generalized outline of rule-generation as a search process. Numerous variations on the common theme can be obtained by making particular choices about how exactly to fill the boxes. For instance, the production of new descriptions from current descriptions may be predominantly from specific to general, or vice versa, or either way. Independently, the evaluation may take account mainly of positive evidence, mainly negative evidence, or both equally.

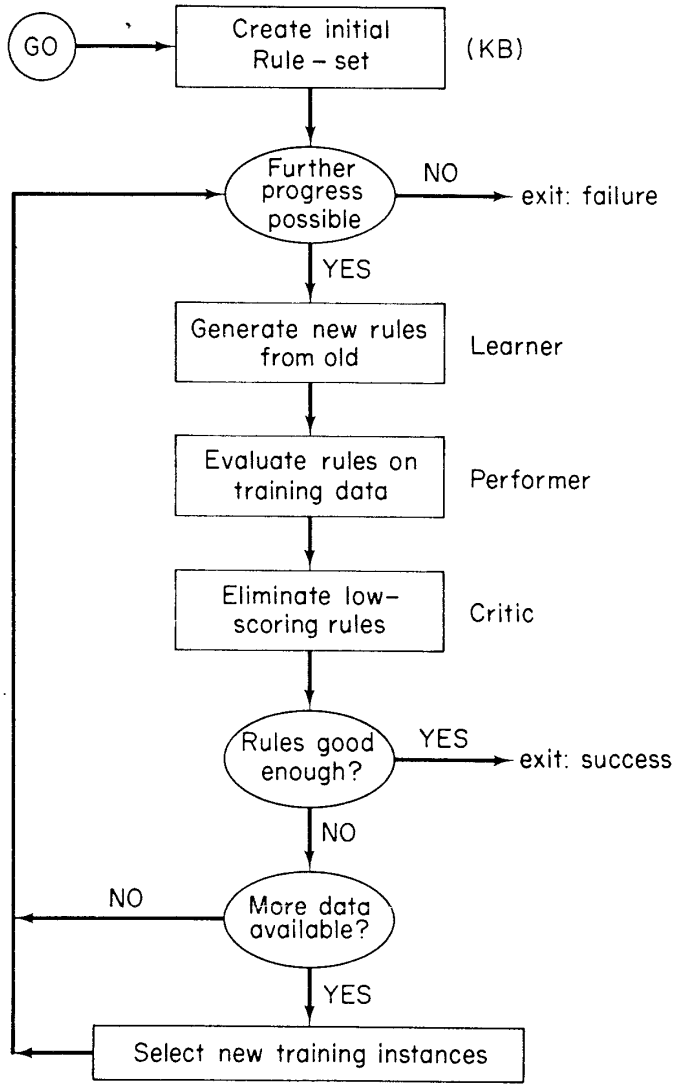

Figure 3.6 -- General rule induction flowchart.
Most differences between learning algorithms depend on the method of generating new rules and the criteria for evaluating successful rules.

Likewise the number of plausible rule structures retained on each cycle may vary. In Quinlan's and Winston's programs, only a single rule is kept. Other systems keep as many as will fit in RAM. In addition, different systems make different choices about how many training items to focus on at any one time. This can range from examining a single instance at a time, through the concept of a window of selected instances, to taking the entire training set.

Finally, of course, description languages differ greatly. The description language limits what can be learned. Even more important, it determines how well people can understand the results of the learning process.

If one single message comes over from the present chapter, it should be this: that these choices are inevitably compromises. You cannot have the best of everything.

You might choose a humanly comprehensible description language and find that it is hard to express the required kinds of concepts in it. You might want a very expressive description language, but find

that it is hard for humans to comprehend and/or it slows down the learning procedure. Not all the desirable attributes of a learning system are mutually compatible. The art of AI in this field (as in others) lies in a successful balancing of conflicting requirements.

3.7 REFERENCES

Buchanan, Bruce (1976) Scientific Theory Formation by Computer: in Simon, J. C. (ed.) Computer Oriented Learning Processes: Noordhoff, Leyden.

Buchanan, B. G. & Feigenbaum, Edward (1978) Dendral and Meta-Dendral: their Applications Dimension: *Artificial Intelligence*, 11.

Buchanan, B. G. & Mitchell, Tom (1978) Model-Directed Learning of Production Rules: in Waterman & Hayes-Roth (eds.) Pattern-Directed Inference Systems: Academic Press, New York.

Dietterich, Thomas & Michalski, Ryszard (1981) Inductive Learning of Structural Description s: *Artificial Intelligence*, 16.

Hart, Anna E. (1985) Experience in the Use of an Inductive System in Knowledge Engineering: in Bramer, Max (ed.) Research & Development in Expert Systems Cambridge University Press.

Hunt, Earl, Marin & Stone (1966) Experiments in Induction: Academic Press, New York.

Langley, Patrick (1977) Rediscovering Physics with Bacon-3: Proc. 5th IJCAI.

Langley, Patrick (1981) Data-driven Discovery of Physical Laws: *Cognitive Science*, 5.

Larson, J. & Michalski, Ryszard (1978) Inductive Inference of VL Decision Rules: in Waterman & Hayes-Roth (eds.) Pattern-Directed Inference Systems: Academic Press, New York.

Lenat, Douglas (1982) The Nature of Heuristics: *Artificial Intelligence*, 19.

Michalski, Ryszard & Chilausky, R. L. (1980) Learning by Being Told and Learning from Examples ...: *Journal of Policy Analysis & Information Systems*, 4.

Michalski, Ryszard & Larson, J. B. (1978) Selection of Most Representative Training Examples & Incremental Generation of VL1 Hypotheses ...: Report 867, University of Illinois, Urbana.

Michalski, Carbonell & Mitchell, (eds.) (1983) Machine Learning: Tioga Press, Palo Alto.

Michie, Donald & Johnston, Rory (1985) The Creative Computer: Pelican Books, Harmondsworth.

Mitchell, Thomas (1982) Generalization as search: *Artificial Intelligence*, 18.

Popper, Karl (1959) The Logic of Scientific Discovery: Basic Books, N.Y.

Quinlan, John Ross (1979) Induction over Large Databases: Report HPP-79-14, Stanford University.

Quinlan, John Ross (1982) Semi-autonomous Acquisition of Pattern-based Knowledge: in Michie, Donald (ed.) Introductory Readings in Expert Systems: Gordon & Breach.

Quinqueton, Joel & Sallentin, Jean (1983) Algorithms for Learning Logical Formulas. Proc. 8th IJCAI, William Kaufmann, California.

Smith, Stephen F. (1983) Flexible Learning of Problem-Solving Heuristics through Adaptive Search: Proc. 8th IJCAI, William Kaufmann, California.

Winston, Patrick (1975) (ed.) The Psychology of Computer Vision: McGraw-Hill, New York.

Winston, Patrick (1984) Artificial Intelligence, 2nd edition: Addison-Wesley, Massachusetts.

4

Evolutionary Learning Strategies

We have seen that some AI scientists take the human brain as a model when they try to design learning systems, but there is another natural model for self-improving systems -- the process of evolution. It is certainly effective as a means of creating ever more advanced organisms, otherwise we would not be here discussing it. Admittedly it is slow; but it can be speeded up in computer simulation. Above all, it is relatively well understood -- better, in many respects, than the inner workings of the human brain. It is simple enough for us to copy with some hope of success.

As a matter of fact this simplicity is partly illusory. Recent discoveries in biochemistry have smudged the stark clarities of the classic Darwinian/Mendelian synthesis. It is obvious that real-life genes have plenty of surprises still in store (Dawkins, 1978). Genetic engineering will complicate the picture still further. Nevertheless the broad outline of the theory remains valid: some organisms survive to breed, others do not; the heritable characteristics of the survivors proliferate. The idea of systems that improve their performance by an analogy (albeit a simplified one) with the evolutionary process is an attractive one; and it has produced some striking results.

The purpose of this chapter is to investigate an evolutionary, or Darwinian, approach to the problem of machine learning. We set this topic apart as a separate subfield in its own right because it has arisen outside mainstream AI and is only now being recognized as a legitimate AI technique, and also because of its distinctive nature.

4.1 THE EVOLUTION OF IDEAS

In an evolutionary learning scheme there is a population of structures which are treated as pseudo-organisms. Each of these structures (which we refer to as 'rules' for compatibility with other writers) defines a potential solution to the problem at hand. They are also used to give rise to new structures (the 'offspring') in ways that mimic some of the features of biological reproduction (Holland, 1975).

Selection of which rules survive longest and have greatest likelihood of 'breeding' depends on their performance at the task in hand -- survival of the fittest.

A typical genetic adaptation algorithm runs as follows.

(1) Generate an initial population of rules at random.
(2) Evaluate the rules and if the overall average is good enough halt and display the best of them.
(3) For each rule compute its selection probability p=e/E where e is its individual score and E is the total score of all the rules.
(4) Generate the next population by selecting according to the selection probabilities and applying certain genetic operators. Repeat from step 2.

Each pass round the loop corresponds to a single generation.

The performance of such an abstract model of competition among organisms depends on a number of factors -- the structure of the rules (description language), the method of evaluation (the critic) and the precise nature of the genetic operators.

The most important genetic operators are crossover, inversion and mutation. Crossover is what happens when two rule structures 'mate' -- chunks of genetic material are exchanged and combined.

Inversion reorders the sequence of components in a rule: this brings together elements that were formerly far apart and separates items that were close together. The importance of inversion lies in its effects on later crossings. When rules are sliced up and recombined, neighbouring elements tend to stay together (as in real genetic material). To explore the space of potential rules thoroughly it is necessary sometimes to part closely linked items and bring distant ones together.

Finally mutation simply involves making a few random changes: a rule is 'zapped' or 'clobbered' in an unpredictable way. The role of mutation is much misunderstood. In genetic algorithms (as in nature) it is a 'background operator'. Its purpose is merely to ensure that the system does not get stuck at a local optimum. The failure of earlier attempts at simulated evolution (e.g. Fogel, Owens & Walsh, 1966) can be largely attributed to their over-reliance on mutation as the sole means of genetic change. In fact crossover is the primary means of generating new structures for testing. Inversion and mutation play ancillary roles.

Our discussion so far has been highly abstract. We may have a clear mental image of what it means for two rabbits or two dogs to mate and produce offspring. But what exactly do we mean when we talk about 'mating' or 'mutating' a rule? An informal example should help to clarify matters.

Below is a list of ten commandments, or rules, that should be familiar to most readers. They are expressed in simplified language for reasons that will become apparent.

Each rule has been put into a standard form

Prohibition/Exhortation + verbal group + object

so that the genetic operators can be applied without undue complication.

Generation 1

(1) Thou shalt not	have	other gods.
(2) Thou shalt not	make	graven images.
(3) Thou shalt not	take in vain	the name of God.
(4) Thou shalt	keep	the sabbath.
(5) Thou shalt	honour	thy father and mother.
(6) Thou shalt not	kill	anyone.
(7) Thou shalt not	commit adultery with	anyone.
(8) Thou shalt not	steal	other people's property.
(9) Thou shalt not	bear false witness	against thy neighbour.
(10) Thou shalt not	covet	thy neighbour's goods.

There are a number of criteria we might use to evaluate such rules. Without wishing to be drawn into profound moral judgements, let us say that the score obtained by a rule depends on how many people obey it. Some will command general assent; others will be disobeyed so frequently that they fall into disrepute.

With this yardstick we can imagine that, after one generation, rules 4, 9 and 10 score poorly. They are deleted and replaced by mating a pair of the surviving rules. For instance

(2) Thou shalt not	make	graven images.

and

(5) Thou shalt	honour	thy father and mother.

might produce as offspring

(2a) Thou shalt not	make	thy father and mother.
(5a) Thou shalt	honour	graven images.

among other possibilities. The next generation, therefore, might resemble the revised list below.

Generation 2

	(1) Thou shalt not	have	other gods.
(mut.)	(2) Thou shalt not	bake	raven images.
	(3) Thou shalt not	take in vain	the name of God.
(7 + 5)	(4) Thou shalt not	commit adultery with	thy father & mother.
(mut.)	(5) Thou shalt not	honour	thy father & mother.
	(6) Thou shalt not	kill	anyone.
	(7) Thou shalt not	commit adultery with	anyone.
	(8) Thou shalt not	steal	other people's property.
(6 + 8)	(9) Thou shalt not	kill	other people's property.
(5 + 2)	(10) Thou shalt	honour	graven images.

Rules 4, 9 and 10 from the first generation have been 'killed off' and replaced by descendants of the survivors. For example the new rule 10 is a combination of old rules 5 and2. In addition, a couple of rules (2 and5) have been subjected to mutations.

This illustration shows that the use of genetic operators imposes constraints on the form of the description language. Ideally it should consist of fixed-length, position-independent strings. Our rules have fixed length (always three components) but are not truly position independent. This restricts the use of inversion, since although

Other gods	thou shalt not	have

makes a kind of sense

Graven images	make	thou shalt

does not. Thus the syntax places restrictions on the rules that can be discovered. It is not clear whether

Thy father and mother	make	other people's property

ought to count as a well-formed rule or not.

However, we have pursued this example far enough. It gives an insight into the genetic operators in action. It also serves to highlight the relationship between genetic algorithms and the traditional 'Monte Carlo' methods as used in some branches of statistics and operational research. Both rely on controlled randomness, but there are important differences.

Genetic algorithms are in effect modified Monte Carlo procedures. Monte Carlo methods are used routinely in operational research for problems that defy analytical solution. These are just the sort of

problems that AI scientists wrestle with in different contexts, yet AI workers in general have made little effort to incorporate such techniques into their own work.

In a 'pure' Monte Carlo method, random solutions are generated for as long as there is time left, and the best solution found during the process is retained. Each trial generates a completely fresh potential solution, randomly, with no reference to what has gone before.

Fig. 4.1 — Baking images of ravens.

Figure 4.1 -- Baking images of ravens.

A pure Monte Carlo method is essentially blind search. A genetic algorithm, on the other hand, makes use of what it finds to direct further search. Specific patterns that contribute to good performance are preserved and propagated throughout the knowledge base (i.e. the population of rules). They are recombined in slightly different contexts. In effect the search is preferentially guided towards regions of the (multi-dimensional) problem space where good results have been found in the past. Unless the evaluation function is extraordinarily discontinuous, one would expect this to lead a reasonable search strategy. (Rada, 1984).

In fact, using the pure Monte Carlo technique as a baseline, we can usefully distinguish four levels of complexity:

(1) Pure Monte Carlo methods;
(2) Mutation-only methods;
(3) Basic genetic algorithms;
(4) The 'elitist' strategy.

Each of these four levels is characterized by a different answer to the question: how is the next candidate to be chosen? Remember that learning can be viewed as a kind of search (Chapter 3). The next candidate is the next structure to be examined in the search through the rule-space. Method (1) simply generates a new structure on each trial, completely at random. It is a genuine 'trial and error' approach. Method (2) modifies existing structures to make new ones for testing. Thus new structures tend to resemble those that have succeeded in the past. Method (3) generates new structures from pairs of previously successful ones. This can be shown to be a more efficient search strategy than Method (2), since it has the effect of searching on more than one dimension of the problem space simultaneously.

There is therefore evidence that these correspond to levels of increasing efficiency. The 'elitist strategy' (Method (4)) is a variant on the evolutionary theme. It differs from the basic genetic algorithm in only one respect: the best rule never dies; it remains till supplanted by a better one. In the algorithm as outlined at the beginning of this section, the whole population is replaced at the end of each cycle. In the example of the ten commandments, on the other hand, some rules were preserved from the first generation to the next.

It is interesting to speculate on the biological analogies. We offer the following tempting, but unproven, historical parallel:

(1) Primeval pre-organic chemistry;
(2) Asexual reproduction;
(3) Sexual reproduction;
(4) Artificial selection.

In this scheme the 'brave new world' of genetic engineering falls under heading (4). It is the sort of thing we have been doing to our plants and animals for a long time. We may shortly begin to practise it on human beings.

4.2 THE POKER.FACED MACHINE

One of the more interesting of practical evolutionary learning systems is LS-1, described by Smith (1980, 1984). This is a general-purpose learning system which can be adapted to a variety of tasks. It has been tested on a maze-walking problem and in the game of Draw Poker.

Its description language, KS-1, is based on the idea of production systems (Davis & King, 1977). Each rule consists of a variable number of fixed-length productions. The individual productions have the following format.

SV-patterns WM-patterns Message Operator

The SV-patterns are patterns that match state variables supplied from the environment. WM-patterns match patterns in the working memory. The message is deposited in working memory when the production is successfully triggered, i.e. when both kinds of patterns are matched. The operator carries out an output operation, i.e. does something in the task domain. The operator can be a NOOP, which does nothing, so some productions only have an internal effect. The patterns are expressed in a simple binary language consisting of the symbols 0, 1 and * of which the last is a *don't care* field. Thus 001* is a pattern that would match a four-character string starting with two zeroes, followed by a one and ending with any other character.

This language forces the input to the system to be expressed as (or translated into) a binary state-vector (a representation we have met before). But notice that the number of productions in a single rule is not fixed. This means that the learning component is in effect generating little programs in a production-rule language. The programs can respond to inputs, leave messages for each other, and generate outputs. Message-passing increases the complexity of the system greatly because it permits productions to cooperate.

The critic employed by LS-1 has to be given some domain-dependent evaluation measures; but it also uses task-independent measures. It assesses structural properties of rules (such as the potential for intercommunication), dynamic properties of rules (derived from tracing execution of the production-rules) and the rule size. Although there is no limit on the number of productions in a rule, the LS-1 critic incorporates a bias towards conciseness. (This can conflict with the desire to have the best possible performance.)

LS-1 succeeded in a relatively challenging task -- learning to play draw poker. In this game five cards are dealt to each player. Then both players alternately have the options of betting, calling or dropping. If a call is made each player may replace up to three cards in his hand with new cards from the deck. A drop terminates the current round. If a second call is made both hands are displayed and the player holding the higher ranking hand collects the money in the pot.

The state of the game is presented to LS-1 in terms of seven integer variables (which are in fact handled as one long bit-string). These are:

VDHAND	value of system's hand
POT	amount in the pot
LASTBET	size of the latest bet
POTBET	ratio of money in pot to last bet
ORPL	number of cards replaced by opponent
BLUFFO	measure of likelihood of bluffing opponent
OSTYLE	measure of opponent's conservatism in play.

LS-1 was pitted against a poker betting program hand-crafted by Waterman (1970) and judged to perform at the level of an experienced human player. No human would have had the patience to play against it long enough! There were four possible decisions for each play: Bet High, Bet Low, Call or Drop.

The objective of the program was to complete 10 consecutive rounds in complete agreement with Waterman's 'axioms' of poker. These axioms permitted the LS-1 critic to examine the record of play for any hand and deduce whether any mistakes had been made, taking into account both hands. The criterion of 10 successive rounds in agreement with the poker axioms was easily achieved. It was too easy, in fact. It turned out that Waterman's program was not suitable for long games of up to 40,000

hands. It tended to judge LS-1, incorrectly, as a very conservative player. This made it susceptible to bluffing. It had to be revised to make it a more formidable opponent.

After tuning the opponent, LS-1 played another 40,000 rounds or so of poker, consuming about 2 DEC-10 CPU hours, and at the end was capable of playing nine successful rounds in a row. (Smith does not reveal which player left the table with fuller pockets!) This, though not reaching the original criterion, was still an impressive level of performance. Its bet decisions agreed with those deduced from the axioms 82% of the time.

LS-1 performed creditably in two different domains, showing that a successful general-purpose genetic learning system is quite feasible. Its main drawback is the obscurity of the KS-1 description language. Nowhere in his thesis does Smith quote any of the knowledge gained by LS-1. This is because it would be unintelligible. In effect, LS-1 is a black-box method because the rules are not intended for public consumption. (See Chapter 2.) This is a problem with genetic algorithms in general. The requirements of the genetic operators (for rules that can be sliced up and stitched together with a minimum of fuss) tend to conflict with the requirement that acquired knowledge should be open to inspection. In nature it appears that there are 'punctuation' segments in the genetic code for precisely this reason: they ensure that cuts occur where they would not make nonsense of the genetic message (most of the time). But of course DNA was not designed for readability, and we are only beginning to understand its description language.

4.3 THE VOYAGE OF THE BEAGLE

BEAGLE (Biological Evolutionary Algorithm Generating Logical Expressions) is an evolutionary learning system that overcomes -- to some extent -- the problem of obscurity in the description language by representing rules as Boolean expressions. These are held internally as tree structures. Its objective is to learn discriminant rules by examining a database of examples, using a method termed 'Naturalistic Selection'.

4.3.1 The BEAGLE has landed

The original BEAGLE system (Forsyth, 1981) had two main modules, but the most recent version consists of six separate programs.

SEED	Selectively Extracts Example Data
ROOT	Rule-Oriented Optimization Tester
HERB	Heuristic Evolutionary Rule Breeder
STEM	Signature Table Evaluation Module
LEAF	Logical Evaluator And Forecaster
PLUM	Procedural Language Utilization Module

A diagram of how they link together appears as Fig. 4.2.

SEED is a simple data extraction program. It interfaces BEAGLE to external databases. It can read datafiles in a few simple formats, including comma-delimited or space-delimited ASCII files as produced by dBase II and other popular packages. It performs one or both of the following functions:

(1) splitting the database in two (random subsets);
(2) appending leading and/or lagging variables for time-series analyses.

The former is useful for making the important distinction between training and test data sets. The latter permits data which is chronologically ordered to be used in various forecasting tasks.

ROOT is merely a preliminary batch-tester for user-supplied rules. This allows the user to suggest initial hunches. (In almost all cases these are quickly superseded by superior machine-made rules.)

HERB is the main module: it actually generates new rules. It takes a datafile from SEED, a tag-file (describing the variables or fields) and an initial rule file as input (which normally comes from ROOT). It produces a new rule file as output. The initial rule file may be empty to start with, except for the 'target expression', which is what you want to predict. For instance, in a weather-forecasting example, a suitable target expression might be

((NEXTRAIN < 0.4) & (NEXTSUN > 4.0)) $

if we were interested in predicting fine days. This expression would be true when the following day's rainfall was below 0.4 (mm) and its sunshine was more than 4.0 (hours).

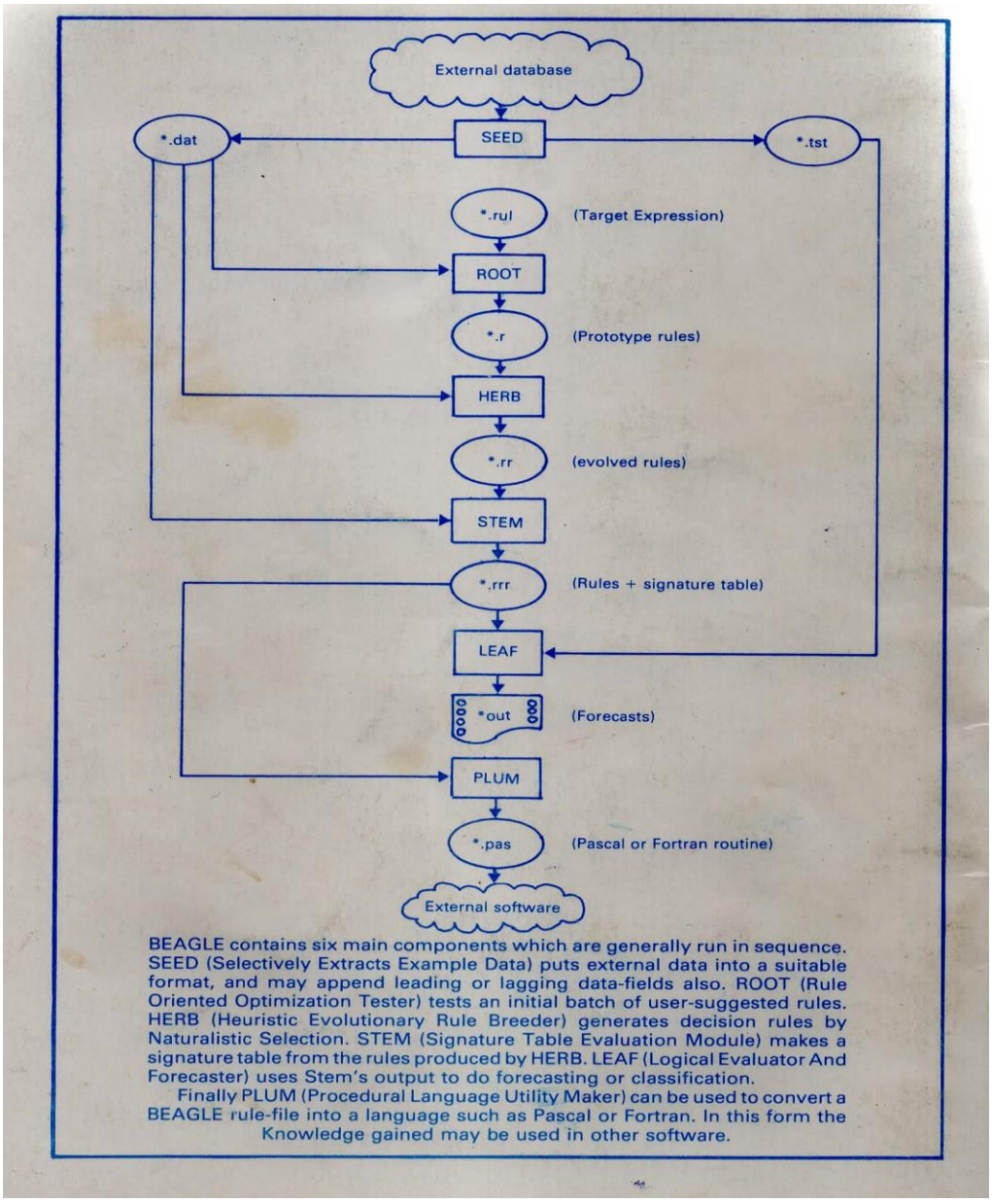

Figure 4.2 -- The BEAGLE has landed.

The rule language allows variable names (such as NEXTSUN) which refer to fields in the database, numeric constants (such as 4.0) and the following operators.

Logical

&	AND
|	Inclusive OR
!	NOT

Relational

=	Equals
<>	Does not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Arithmetic

+	Addition
-	Subtraction
*	Multiplication

These are combined into fully bracketed Boolean expressions, terminated by a dollar sign ($). There is no operator procedence, so brackets are mandatory. If necessary, numeric values are coerced to logical values by the convention that zero and negative numbers are false and numbers greater than zero are true. (Character string fields can be handled, by a kind of phonetic hashing, but the package is designed primarily for numeric data.)

Note that the target expression and the rules are encoded in the same language. This means there is no need for a special category field. BEAGLE can be used with whatever dependent variable seems appropriate or (as above) with a target that is the logical combination of dependent variables. For instance it is just as easy to try forecasting the wind as the rainfall or sunshine, provided the data is available.

HPRB works by running through the following procedure, expressed here in 'pidgin pascal'. (This is in fact a simplified outline of the HERB main program.)

```
REPEAT
        mainloop := mainloop + 1;
        runs := 0;
        REPEAT
                runs := runs + 1;
                reset (datafllle); (* start at beginning of data *)
                FOR nx := 1 TO samples DO (* for every training instance *)
                        (* Get the next data sample *)
                        (* Try all the rules on it *)
                        ;
                scoring; (* evaluate new rules' performance *)
                culling; (* get rid of sub-standard rules *)
                eugenics; (* mate the good ones *)
                mutation; (* make a few random changes *)
```

```
            tidying; (* clean up the mess *)
            UNTIL (runs >= gens); (* enough generations *)
        savebest; (* keep the top rule *)
        wipeout; (* eliminate its variables from further use *)
        UNTIL (mainloop >= maxloops) OR (varsleft < 2);
(* dump top rules on file *)
```

The main subprograms are explained in comments between '(*' and '*)' but there are one or two points that need further clarification.

Mating is achieved by picking out a random subtree (or subexpression) from each of two high-scoring rules and tying them together with a randomly chosen connective. Thus from the two parent rules

```
        (LASTWIND > (LASTRATN + 5.80))
and
        ((RAINFALL * 8.50) = RAINFALL)
```

possible descendants could be

```
        (LASTWIND > 8.50)
        ((LASTRAIN + 5.80) = RAINFALL)
```

and so on. Note the rule ((RAINFALL * 8.50) = RAINFALL) can be true when RAINFALL=0. BEAGLE does not always find the simplest way to say what it means.

The scoring procedure rests ultimately on the Chi-squared statistic. That is to say, each rule can give a true or false result and the target expression can also yield a true or false result. Thus each joint outcome falls into one cell of a fourfold contingency table. The more this table departs from chance expectation, the better -- i.e. the more effectively the target can be predicted from the rule value. In evaluating a rule the raw score, based on Chi-squared, is biased by subtracting the logarithm (or optionally the square root) of the rule's size to give a pressure towards short rules.

The tidying routine cleans up any solecisms introduced by the mutation procedure and performs a number of syntactic manipulations intended to simplify the rule and make it more comprehensible. Thus

```
        (RAINFALL + -2.5)
```

would become

```
        (RAINFALL - 2.5)
```

and

```
        ((10.75 + 99.05) & (WINDGUST < 30))
```

would be reduced to

```
        (WINDGUST < 30)
```

since 10.75 + 99.05 = 109.80 which counts as true in a logical context.

One point worth noting is that HERB contains two nested loops. The inner loop goes through a given number of generations in the manner of a conventional genetic algorithm, using the elitist strategy. At the end of this process the best rule found so far is retained and all the variables it uses are disqualified from further use. Then the process repeats. Thus several rules are generated for output, each one being the best that was found with the variables remaining. Earlier versions of BEAGLE just came up with a single 'best' rule. This was not found to be generally satisfactory. The newer system makes better use of information inherent in all the problem variables.

STEM takes the rules dumped by HERB and uses them to construct a signature table. (See Chapter 2.) Each rule is a predicate that can be in one of two states -- true or false. Therefore with, say, four rules there are 16 possible states for the four rules in combination. Each of these 16 combinations defines a particular 'signature' (or 'fingerprint'). STEM re-examines the training data and counts the number of times each signature occurs; at the same time it accumulates the average value of the target expression for each signature. STEM will also optionally, drop the least useful rule the one whose absence would least degrade its prediction score.

The revised rule-file (with signature table) produced by STEM is in a form that LEAF can use for forecasting. LEAF simply runs over a fresh data set (which uses the same variables as the training data, or at least all the variables mentioned in the rules) and estimates what the value of the target expression should be. It does this by evaluating all the rules and using them to point to a cell in the signature table. LEAF's output is sorted in descending order of the estimated target value. If the actual target value is known, LEAF also works out the success rate of the rules. In principle it could handle missing rule values by looking at all the cells corresponding to the values that are not missing, but it does not do so at present.

Finally PLUM realizes the objective of programming by example. It translates a BEAGLE rule-file into a Pascal procedure or Fortran subroutine. This means that once you have found a good decision rule with BEAGLE's help, you are free to export it into other software.

4.3.2 Empirical testing

BEAGLE has been tested in various domains including categorizing cardiac patients, forecasting commodity prices, analyzing drilling data from North Sea boreholes and gambling. It is suitable for a wide range of diagnostic classification tasks. Here we present some results from a medical and a meteorological trial.

In the medical example, it was used to find a set of rules that could discriminate heavy from light drinkers on the basis of some biochemical tests. The data used were for 345 men, measured on six variables. This data was kindly supplied by BUPA Medical Research Ltd. (See Appendix A.) The six variables were as follows.

MCV	Mean Corpuscular Volume
ALKPHOS	Alkaline Phosphotase
SGPT	Alanine Aminotransferase
SGOT	Aspartate Aminotransferase
GAMMAGT	Gamma-Glutamyl Transpeptidase
DRINKS	the number of half-pint-equivalents of alcoholic beverages drunk per day

By way of brief explanation, MCV measures the mean size of red blood cells; ALKPHOS is an enzyme created in the bone marrow and the liver whose level is known to change in women after the menopause; SGPT is an enzyme in the blood associated with liver function in so far as raised values are found when the liver is damaged; SGOT is another enzyme indicative of liver function whose value is raised by toxic hepatitis, malignancies and so on; GAMMAGT is a sensitive indicator of cellular liver damage used to assess alcohol abuse. Thus the first five variables are blood tests which are thought to be sensitive to liver or blood disorders arising from excessive alcohol consumption.

With the target expression (DRINKS < 7) after a run of 25 generations on a training set of 113 cases, BEAGLE found the following rules. The numbers of following each rule correspond to the contingency table as follows.

Target: (DRINKS < 7)
Rule 1: ((MCV >= 91) < (20 >= GAMMAGT))

	Target-True	Target-False
Rule-True	80	7
Rule-False	11	15

```
      ( DRINKS < 7.00)$
(( MCV >= 91.00) <= (20.00 >= GAMMAGT ))
$     80    7     11    15
( SGPT >= 44.83)
$     85    14    5     8
( SGOT >= 33.39)
$     89    16    1     6
( ) $
       0.8053          113
000    0.0000            4
001    1.0000            4
010    0.0000            1
011   10.0000           17
100    0.0000            1
101    4.0000            4
110    1.0000            1
111   75.0000           81
```

Here again HERB has not expressed it as we might like, even after the tidying procedure. We would probably prefer

((MCV < 91) I (GAMMAGT < 20))

and in the long run it should be possible to get the system to reformulate rules to correspond to the (implicit) syntax we impose on arithmetical and logical expressions.

This rule set gave 87% correct classification on the 113 training cases. More significantly it was correct on 85% of the 232 test cases which it had not previously encountered. This compares favourably with a run of the well-known SPSS package, using its discriminant analysis module on the same training and test data. The SPSS linear discriminant functions achieved 77.5% success on unseen data. The signature-table format (and the rules themselves) allow for non-linear interactions between variables; and in this case it would seem to be an advantage.

A disadvantage, however, is illustrated here too. Only two of the eight cells in the signature table have a sample size running into double figures. This is a problem we mentioned in Chapter 3, in

connection with ID3 decision trees. The table is in effect a sparse matrix. Really only two of the cells can be relied upon for prediction: the rest contain too few examples. For this reason LEAF flags forecasts based on small samples with a warning. It may prove better, however, simply to admit ignorance in these cases. What the system has really learnt here (from a training set that is smaller than desirable) is that if all three rules are true there is a 75/81 chance of the target being true, and if the first one is false and the other two true there is a 10/17 chance. In other cases it simply does not know. Even this one-sided knowledge could be useful for screening purposes. People in the all-Yes group are unlikely to have a drinking problem.

For the weather-forecasting test, BEAGLE was run on a training set of London readings for March, April and May of 1983 and 1984. It was then tested on the 90 days of March, April and May 1985 (excluding 1st March and 31st May on account of missing preceding and following data) together with 28 days from February 1984. The object of the excercise was to predict rain on the foilowing day of more than 0.1mm. The rule set produced after a 32-generation run was as follows.

```
      ( NEXTRAIN > 0.1000)$
( RAINFALL > 0.00)
$     75    45    14    48
(( SUNSHINE < WINDMEAN) > (( LASTRAIN < 0.300) + (WINDGUST >= 45.00)))
$     50    18    39    75
(( PRESSURE - 997.00) < MAXTEMP)
$     60    23    29    70
( ) $
       0.4890          182
000    6.0000           44
001    4.0000           10
010    2.0000            4
011    2.0000            4
100   13.0000           37
101   16.0000           23
110    8.0000           14
111   38.0000           46
```

Note that BEAGLE arranges for all the rules to point in the same direction, i.e. to be favourable to the target when true. To read the signature table you need to know that the first rule sets the first bit in the signature and the final rule sets the last bit. So the signature 011 points to cases where rule 1 was false but rules 2 and 3 were true.

Sixteen variables were used in this example.

DATE	The date
MINTEMP	Minimum temperature in degrees Celsius
MAXTEMP	Maximum temperature in degrees Celsius
MORNDAMP	Humidity % at 0600 hours
EVEDAMP	Humidity % at 1500 hours
RAINFALL	Rainfall in millimetres
WINDMEAN	Average wind speed in knots
WINDGUST	Maximum wind gust in knots
SUNSHINE	Hours of sunshine
PRESSURE	Atmospheric Pressure in millibars at 1800 hours (sometimes 1900 hours)
LASTSUN	Previous day's SUNSHINE
NEXTSUN	Following day's SUNSHINE
LASTRAIN	Previous day's RAINFALL
NEXTRAIN	Following day's RAINFALL

LASTWIND Previous day's WINDMEAN
NEXTWIND Following day's WINDMEAN

All data are London readings. A single example, from May-85, is

'30-MAY', 7.5,19.6,70,24,0.0,13.2,29,15.5,1027,15.2,11.9, 0.0, 0.0,10.6,12.7

which, as it happens, was the sunniest May day in central London since records began in1929.

On the unknown data from spring of 1985 the success rate was 68%, which is quite respectable for an amateur. Chance expectation would be close to 50%. (A LEAF printout is shown in Appendix B.) Britain is positioned at the edge of a continent and an ocean, lying in the track of Atlantic depressions. In consequence, the weather of the British Isles is notoriously upredictable. The Meteorological Office claims to get 80% of its forecasts right; but these require enormous computing resources, and are usually couched in rather vague terms. If you require specific predictions, such as whether there will be more than 6 hours of sunshine tomorrow in a particular location, the success rate falls (and the cost rises!).

In fact BEAGLE's performance on unknown data was better than it appears. By ignoring the forecasts that LEAF flagged as based on too small a sample in the signature table, the overall success rate rises to 65/92 or 70.65% (with 26 unknowns). This uses only the four groups with the biggest samples. Whether this conservative strategy confers any practical benefit depends on your application; but in any case it is useful that the system can confess when it is uncertain. If you must forecast the weather every day, you just have to make the best prediction you can. But if your task is picking horses to win races or buying shares when they are likely to go up, you may be able to afford to wait for a confident forecast -- and then strike!

The results look even better whey you add up the rainfall for the 37 days when BEAGLE was confident of rain (75.8mm in total) and the 55 days when it confidently forecase a dry day (22.8mm). For the former the average daily rainfall was 2.0486mm, while for the latter it was 0.4145mm. Clearly it is on to something.

But this is starting to sound like special pleading. Suffice it to say that there is evidence that combining on evolutionary learning strategy with a signature table works quite well.

(Commercial versions of the BEAGLE package are available for VAX minicomputers and for PC-compatible microcomputers from Warm Boot Ltd.)

4.4 CONCLUSIONS

We have used the term Naturalistic Selection to describe a variety of methods that draw on evolutionary principles. This covers systems we have not described here (see Lenat, 1983, for example). Naturalistic Selection is no panacea, but it is a robust and practical technique in cases where there is a very large search space. If your problem has

(1) information structures that encode potential solutions,
(2) a way of evaluating those structures,
(3) formal means of chopping up and re-splicing those structures without creating nonsense rules,
(4) no obvious algorithmic solution,

then you would be well advised to consider an evolutionary learning scheme. After all, there are not many computing techniques that have proved their worth over 3 billion years of field testing.

4.5 REFERENCES

Davis, R. & King, J. (1977) An Overview of Production Systems: in Elcock & Michie (eds.) Machine Intelligence, B: Ellis Horwood, Chichester.

Dawkins, Richard (1978) The Selfish Gene: Granada, St. Albans.

Fogel, Owens & Walsh (1966) Artificial Intelligence through Simulated Evolution: Wiley.

Forsyth, Richard (1981) BEAGLE: A Darwinian Approach to pattern Recognition : *Kybernetes*, 10.

Holland, John (1975) Adaptation in Natural and Artificial systems: University of Michigan Press.

Lenat, Douglas (1983) The Role of Heuristics in Learning by Discovery: in Michalski et al. (eds.) *Machine Learning*: Tioga Press.

Rada, Roy (1984) Automating Knowledge Acquistion: in Forsyth, R. S. (ed.) *Expert Systems, Principles and Case Studies*: Chapman & Hall, London.

Smith, Stephen F. (1980) A Learning System Based on Genetic Adaptive Algorithms: PhD Thesis, Dept. Computer Science, University of Pittsburgh.

Smith, Stephen F. (1984) Adaptive Learning Systems: in Forsyth, R. S. (ed.) *Expert Systems: Principles and Case Studies*: Chapman & Hall, London.

Waterman, D. A. (1970) Generalization Learning Techniques for Automating the Learning of Heuristics: *Artificial Intelligence*, 1.

5

Towards the learning machine

In Chapter 1 we identified four key components of any learning system -- the Performer, the Learner, the Rules and the Critic. The performer is necessarily specific to a given task, but the other three can and should be largely domain-independent. We have looked at a number of learning algorithms (Learners) employing a variety of description languages (Rules). We have been less forthcoming about the evaluation functions used (Critics) because the precise form of evaluation depends on the aim of the learning exercise and is thus at the discretion of the user. However, all learning systems must have some kind of performance measure and this nearly always gives some weight to features of the rules other than their effectiveness at the chosen task -- e.g. brevity or structural simplicity.

In this chapter we attempt to draw some of the disparate threads together so that the reader with a practical orientation can gain some advice on how to design a workable learning system (and how not to).

5.1 ASPECTS OF MACHINE LEARNING

Machine learning is potentially applicable to a bewildering diversity of tasks. This means that there are many ways of looking at the problem, each perspective emphasizing some aspects of the overall process at the expense of others. For example

Automatic Programming
Artistic Creativity
Data Compression
Knowledge Synthesis
Optimization
Search
Theory Formation

are all facets of machine learning.

5.1.1 Automatic programming

The objective of automatic programming is for the computer to write its own programs. Typically it is given examples of input-output pairings and has to synthesize a program that will produce the appropriate output for every input. In one sense Smith's LS-1 (Chapter 4) is an automatic programming system. If the description language is a programming language then learning descriptions in that language is a kind of automatic programming.

Automatic programming is a very difficult problem. One reason for this is that we expect our programs to give the right answers under all conditions. (They may not, but we still expect it.) This means that there is no place for noise in the system.

5.1.2 Learning as creativity

The association between learning and creativity is also very strong. Very often learning, in people, is a creative act: we discover how to tie our shoelaces or draw a picture of a cat. We may think we have done something original, but it is not called creativity either because the subject matter is too mundane or because someone else has made the discovery already, or both. Nevertheless the two

activities are very closely related. Learning is creativity at second hand. Conversely, creativity is learning something for the first time ever.

5.1.3 Data compression

Data compression is normally an unintended by-product of machine learning simply because the rules learned are typically much shorter than the training data. Yet they can be used to reconstruct the important properties of that training data. Sometimes, however, data compression is a major objective. Indeed many approaches based on signature tables are best viewed as attempts to reduce a large collection of examples to a compact tabular format.

5.1.4 Knowledge synthesis

Designers of expert systems tend to think of machine learning in terms of synthesizing knowledge. For them it is a short-cut on the long and tortuous road towards obtaining knowledge from a human expert. Currently such knowledge has to be elicited, codified and checked by a knowledge engineer. This involves much human labour. In future much of it may be done by machine.

This is an approach we have stressed in the present work, but it is only one way of looking at the subject.

5.1.5 Learning as optimization

Machine learning can also be seen as a kind of optimization. This is particularly clear in parameter-adjustment systems (like the Perceptron) where the knowledge is expressed as a linear function of some sort. Here the objective is to find a set of parameters or coefficients. When the expression is a simple one this can be achieved by conventional optimization techniques; but a complex expression may not yield to conventional optimization techniques. In such a case, a search is required.

5.1.6 Learning by searching

The search space in a parameter-optimization problem is the set of all possible parameter combinations. Since this space is enormous, efficient heuristics have to be used to explore that space. This is the point we emphasized in Chapter 3, where we considered learning as a search process.

5.1.7 Computer science and computerized science

Finally, the search for new knowledge can be regarded as a process of theory formulation. This is exemplified in those systems that base their methods on induction as practised by the scientific community, past and present (e.g. Bacon the man, Chapter 1, and Bacon the program, Chapter 3).

In the long run this may prove the most significant aspect of machine learning. Scientists who are quite willing to accept the computer as a workhorse for calculations may have reservations about its role as a fellow voyager on the great ocean of truth -- especially if it sails rapidly over the horizon of human comprehension. It is bad enough for a machine to make scientific discoveries that nobody ever thought of: that has already happened to a limited extent. But in due course computers may well discover facts that are literally beyond our understanding. This could be rather humiliating. (In the physics of subatomic particles we have almost reached this sorry state of affairs without much help from the computer.)

This then is the long-term outlook for machine learning: the automation of science itself. It is both a promise and a threat.

5.2 CREATIVE COMPUTERS

The automation of scientific discovery may seem a distant prospect, but the first steps have already been taken. One of the pioneers in this field is Doug Lenat of Stanford University, who devised the Eurisko program (Lenat, 1982).

Eurisko is a discovery program which we have already mentioned briefly. It has been applied in several domains ranging from a naval wargame to the design of VLSI (Very Large Scale Integration) circuits. It begins with a collection of heuristic rules and concepts and applies them to the chosen domain. Its novelty lies in the fact that it can modify its own heuristics, which are expressed like everything else in the system as frame-like data structures. This gives the system great power, since it can adapt and specialize its general rules to deal with new situations.

It has made one discovery for which a patent was subsequently granted. This was a new design for a three-dimensional logic circuit (a NAND/OR gate), which nobody in Silicon Valley -- or anywhere else -- had thought of. Yet it was generated by the application of a single rule.

When Eurisko was applied to VLSI design it already had a heuristic rule, carried over from previous tasks, that said, in effect: if a concept is interesting, try to make it more symmetrical. Applied to a two-dimensional device it led to a more symmetrical version of that device -- which has recently been successfully fabricated. This three-dimensional device can be packed more densely, thus offering greater capacity.

[....]

Figure 5.1 -- 2D and 3D VLSI gates.

> When Eurisko took the logic-gate structure above and applied its heuristic rule about symmetry, it transformed it into a new 3-dimensional structure below. Subsequently a patent was granted for a device based on the lower design, so it represents a genuine non-trivial discovery.

Creativity is seen as the pinnacle of human intelligence, and shrouded in mysterious explanations involving intuition and insight; but Eurisko should give us pause for thought. Here is an example of a genuine discovery resulting from the application of a single rule. Perhaps scientists, as scientists, behave in ways that are simpler than previously thought. They take an existing concept and give it a tweak; or take two old concepts and find a way of joining them. In effect they say to themselves such things as the following.

> Let's try inverting that function.
> I wonder what happens in extreme cases.
> Surely this design can be made more symmetrical?
> These two functions look interesting, can we link them?

These are the sort of operations that Eurisko can carry out. They are also the kind of operations that have been identified as fundamental to scientific creativity by Koestler (1964) and others.

The creative computer is closer than many people realize.

5.3 PROBLEMS OF MACHINE LEARNING

This brings us to a consideration of the problems of machine learning. For the present the main problem is getting it to work at all; but in the longer term one can envisage that the chief problem may not be failure but success -- at least success of a certain kind.

Learning can be a dangerous activity, as the story of the Tree of Knowledge illustrated. Inductively derived knowledge can never be proved, which is why (as we saw in Chapter 1) it has always caused disquiet among philosophers. Yet computerized learning systems indulge in rampant inductionism. This is bound to introduce uncertainties. The trouble is that we respect computer solutions too much. In many cases we are overawed by printout, as if it were the word of God carved in stone.

Suppose a computer system has learned how to identify early signs of leukaemia (using, let us say, standard biochemical tests). Your eldest child is sorted into a high-risk group, and the doctors recommend a course of preventive therapy which includes a potentially hazardous bone-marrow transfusion. Who is in a position to evaluate the risks? Probably neither you nor your doctor, unless the computer-generated rules are more explicit than anything we have seen so far.

Then again, suppose your younger child slips through the screening tests, but later develops the disease. Is the doctor to blame? Is the programmer to blame? Is it anyone's fault? Who, after all, will dare to countermand the computer, even when it is wrong?

Dilemmas like these make it likely that we shall have to certify and ratify computer-produced knowledge. In medicine (and some other fields) decision-rules generated by computers may have to be scrutinized and passed by a standards board before being released for general use, rather as drugs have to be clinically tested. The knowledge, even when released, may need to carry a grading certificate which states the circumstances in which it can safely be used.

The more efficient computers become at inducing new knowledge, the more widely that knowledge will be applied, even in matters of life and death. It is essential that such knowledge be open to inspection. This means that designers of learning systems have a public duty to use comprehensible description languages -- even if that means sacrificing performance. Otherwise we run the risk of generating truly 'unknowable knowledge'. It is quite easy to think of codes that are effective in guiding a machine's decision making but which are virtually incomprehensible to people. Binary machine code is a good example.

But even if the knowledge generated by the next generation of learning systems is intelligible to people, there is no guarantee that they are going to like it when they see it, nor even that the people to whom it matters will be allowed to see it at all. Suppose that a large company applied an inductive rule-finder to an archive of data on past and present employees, and came up with a rule that said in plain English: do not hire any coloured people?

We are already familiar with abuses of credit-rating data-processing systems and the like. The advent of effective machine learning will broaden the scope for such abuses. The problem becomes particularly acute in military applications.

We can easily imagine systems that learn to recognize submarines from sonar signals or aircraft from radar images. It is not too far-fetched to imagine a roving robot with a machine-gun that could be taught how to distinguish English speakers from Russian speakers. We do not have to use our imaginations at all to realize what such machines will do when you open your mouth. So-called 'smart weapons' pose frightening ethical problems for all AI workers, not just those concerned with machine learning. But, as we have argued, learning is fundamental to intelligence; so the problems are particularly severe in this area. Setting loose on the battlefield weapons that are able to learn may be one of the biggest mistakes mankind has ever made. It could also be one of the last.

5.4 THE WAY AHEAD

In any field of human endeavour it is wise to be aware of potential long-term hazards, in order to be better equipped to combat them when the time comes. Machine learning is no exception, although some AI researchers exclude questions of morality from their work (despite the example of physicists and the atomic bomb). That is why we have raised such issues here.

However, the gloom of undiluted pessimism has little value, so let us end this part of the book on a more positive and more practical note, with some advice.

If you wish to build a learning system of your own, here are some hints and tips. They have to be rather generalized, but should still prove a useful starting point for your own researches.

Aims
(1) If the purpose of the project is morally reprehensible (e.g. killing civilians), stop right there.
Description language
(2) Ensure that the description language is capable of expressing the distinctions that will be needed. For instance, if numeric comparisons are required, do not be satisfied with a simple string-matching language.
(3) Try to make the format of the rules as legible as possible for people. Vectors of floating-point numbers are particularly poor in this respect.
Learner
(4) Choose the simplest algorithm you can get away with. There are plenty to choose from in this book. Others can be looked up via the Bibliography.
(5) If you intend to devise a new method of your own, look closely first at natural models (e.g. genetics, the brain, the immune system, science as a social phenomenon, etc.) before designing your algorithms.
Critic
(6) Think hard about the criteria for differentiating successful from unsuccessful trials. If you want the system to learn multiple cooperating rules (as in automatic programming), the problem of credit assignment is especially hard. One possible solution, not fully exploited, is money (computer-based credit, of course, not hard cash). Each rule is rewarded for contributing to successes and penalized for contributing to failures. Rules have to pay each other for services rendered, such as providing information. A quasi-economic model along these lines -- the 'bucket-brigade algorithm' -- has been proposed by John Holland at the University of Michigan (see Lindsay, 1985). But always remember: computers give you what you ask for, not what you want; and machine learning is no exception.
(7) Make sure the evaluation function is modifiable, so that you can vary the weights (e.g. between false positives and false negatives) if necessary; and keep the option of evaluating the rules partly on structural criteria (such as brevity) as well as performance in the task domain. You never know exactly how to evaluate what you want till you see what you get.
General
(8) As far as possible, only ask the machine to learn what it nearly knows already. Machine learning works best when it is tuning or refining existing knowledge. To put it another way: you will get best results by selecting appropriate input variables, by presenting carefully chosen training examples and by fitting the description language to the task in hand. Natural language rules will look different from computer-vision rules. It is not 'cheating' to reduce the search space to manageable proportions before letting the learning program wander round it: it is good sense. The machine's task is then merely to home in on the precise behaviour required (Forsyth & Naylor, 1985).

But the most important advice of all is to give it a try. Machine learning, as we have attempted to show, is not a black art; and there are few fields within computing where it cannot profitably be applied. A number of important ones spring to mind:

Medical diagnosis
Predicting earthquakes
Scientific gambling
Mineral exploration
Game playing
Image recognition
Robotics
Signature verification
Weather forecasting
Intelligent information retrieval.

A full list would occupy too much paper, because learning is the key to intelligence and intelligence is the key to the next generation of software. As in nature, so in programming, the slogan is: adapt or die. You are encouraged to think of ways of making your own programs more adaptable, perhaps using techniques outlined in this book.

With these recommendations the general introduction to our book is finished. In Part 2 we try to practise what we preach by focusing on the last item in the above list -- intelligent information retrieval. This complements our wide-ranging survey by delving more deeply into a single problem domain and showing how machine learning methods fit into an overall problem-solving strategy.

5.5 REFERENCES

Forsvth, Richard & Naylor, Chris (1985) The Hitch-hiker's Guide to Artificial Intelligence: Methuen, London.
Koestler, Arthur (1964) The Act of Creation: Hutchinson (Pan Books) London.
Lenat, Douglas (1982) The Nature of Heuristics: *Artificial Intelligence*, 19.
Lindsay, Robert (1985) Artificial Intelligence Research at the University of Michigan: *AI Magazine*, 6, Summer.