Department of Artificial Intelligence
University of Edinburgh

MACHINE LEARNING

Outline Lecture Notes,
Spring Term 1997

©Chris Mellish, 1997

DAI TEACHING PAPER No. 10

Introduction

This document provides an outline set of lecture notes for the AI3/4 Machine Learning module in 1997. The content of the module is defined by the combination of what is covered in the lectures and what is covered in these notes. Although there is a very heavy overlap, some material (in particular, motivatory remarks, examples and pictures) in the lectures is not yet covered in these notes. This means that, although most of the "facts" are included here, it is still necessary to attend the lecture in order to get the right intuitions for how they fit together. Each Chapter of the notes gives a list of sources where more information about the topic can be obtained. I will frequently refer to the following books:

- Thornton, C. J., *Techniques in Computational Learning*, Chapman and Hall, 1992.
- Carbonell, J., *Machine Learning: Paradigms and Methods*, MIT Press, 1989.
- Shavlik, Jude W. and Dietterich, T., *Readings in Machine Learning*, Morgan Kaufmann, 1990.

Computing Preliminaries

These notes, and the specifications of associated practical work, will often refer to to example code or data in the directory `$ml` (or subdirectories of that). To access this software in the way described, make sure that your `.bashrc` file includes the following line:

```
export ml=~dai/docs/teaching/modules/ml
```

Contents

1 Machine Learning - Overview 9
- 1.1 A Definition 9
- 1.2 Some Overall Comments 9
- 1.3 Parameters of a Learning System 10
  - 1.3.1 Domain/ Knowledge Representation 10
  - 1.3.2 Application type 10
  - 1.3.3 Type of Input 11
  - 1.3.4 Types of Interaction 12
- 1.4 Some Views of Learning 12
  - 1.4.1 Learning as Reinforcement 12
  - 1.4.2 Learning as Search 13
  - 1.4.3 Learning as Optimisation 13
  - 1.4.4 Learning as Curve Fitting 13
- 1.5 Applications of Machine Learning 14
- 1.6 The ML Module 14
- 1.7 Reading 15

2 Concept Learning - Description Spaces 17
- 2.1 Types of Observations 17
- 2.2 Types of Descriptions/Concepts 18
- 2.3 Abstract Characterisation of Description Spaces 19
- 2.4 Examples of Description Spaces 21
  - 2.4.1 Nominal Features 21
  - 2.4.2 Features with ordered values 22
  - 2.4.3 Structured Features 23
  - 2.4.4 Bundles of Independent Features 23

3 Concept Learning - Search Algorithms 25
- 3.1 Search Strategies for Concept Learning 25
- 3.2 Version Spaces 27
- 3.3 The Candidate Elimination Algorithm 27
  - 3.3.1 Pruning a Version Space Representation 27
  - 3.3.2 Applying a Version Space Representation 28

- 3.3.3 Dealing with a Positive Example 28
- 3.3.4 Dealing with a Negative Example 28
- 3.3.5 The Algorithm 28
- 3.4 Disjunctive Descriptions 28
  - 3.4.1 AQ 29
- 3.5 Reading 30

4 Inductive Logic Programming 1 **31**
- 4.1 The Problem 31
- 4.2 Architecture of MIS 32
- 4.3 New Positive Examples 33
- 4.4 Refinement Operators 34
- 4.5 New Negative Examples 35
- 4.6 Search 36
- 4.7 Performance and Conclusions 36
- 4.8 Reading 37

5 Inductive Logic Programming 2 **39**
- 5.1 Improving the Search - Quinlan's FOIL 39
  - 5.1.1 Basic Characteristics 39
  - 5.1.2 Top-Level Algorithm 40
  - 5.1.3 Constructing a Clause 40
  - 5.1.4 Selecting a New Literal 40
  - 5.1.5 Performance and Problems 41
- 5.2 Top-Down and Bottom-Up Methods 41
- 5.3 Inverting Resolution - CIGOL 42
  - 5.3.1 The V Operator 42
  - 5.3.2 The W Operator 43
  - 5.3.3 Search 45
- 5.4 References 45

6 Classification Learning **47**
- 6.1 Algorithms for Classification 47
- 6.2 Demonstration: The 'Animals' Program 48
- 6.3 Numerical Approaches to Classification 48
- 6.4 Reading 49

7 Distance-based Models **51**
- 7.1 Distance Measures 51
- 7.2 Nearest neighbour classification 51
- 7.3 Case/Instance-Based Learning (CBL) 52
  - 7.3.1 Distance Measures 52
  - 7.3.2 Refinements 53

7.3.3 Evaluation . . . 54
7.4 Case Based Reasoning (CBR) . . . 54
7.5 Background Reading . . . 55

8 Bayesian Classification 57
8.1 Useful Statistical Matrices and Vectors . . . 57
8.2 Statistical approaches to generalisation . . . 58
8.3 Example: Multivariate Normal Distribution . . . 59
8.4 Using Statistical “distance” for classification . . . 60
8.5 Bayesian classification . . . 60
8.6 Advantages and Weaknesses of Mathematical and Statistical Techniques . . . 61
8.7 Background Reading . . . 61

9 Information Theory 63
9.1 Basic Introduction to Information Theory . . . 63
9.2 Entropy . . . 64
9.3 Classification and Information . . . 65
9.4 References . . . 65

10 ID3 67
10.1 Decision Trees . . . 67
10.2 CLS . . . 68
10.3 ID3 . . . 69
10.3.1 The Information Theoretic Heuristic . . . 69
10.3.2 Windowing . . . 70
10.4 Some Limitations of ID3 . . . 70
10.5 References . . . 70

11 Refinements on ID3 71
11.1 The Gain Ratio Criterion . . . 71
11.2 Continuous Attributes . . . 71
11.3 Unknown Values . . . 72
11.3.1 Evaluating tests . . . 72
11.3.2 Partitioning the training set . . . 72
11.3.3 Classifying an unseen case . . . 73
11.4 Pruning . . . 73
11.5 Converting to Rules . . . 74
11.6 Windowing . . . 74
11.7 Grouping Attribute Values . . . 74
11.8 Comparison with other approaches . . . 75
11.9 Reading . . . 76

12 Reinforcement Learning 77
12.1 Demonstration: Noughts and Crosses 77
12.2 Reinforcement and Mathematical approaches to generalisation . . 77
12.3 Gradient Descent 78
12.4 Batch vs Incremental Learning 80
12.5 Background Reading 80

13 Linear Classifiers and the Perceptron 81
13.1 Linear classification 81
13.2 The Perceptron Convergence Procedure 82
13.3 The Perceptron 83
13.4 Example: Assigning Roles in Sentences 83
    13.4.1 The Task 83
    13.4.2 Network 83
    13.4.3 Results 84
13.5 Limitations of Perceptrons 84
13.6 Some Reflections on Connectionist Learning 85
13.7 Background Reading 86

14 Explanation Based Generalisation (EBG) 87
14.1 Demonstration: Finger 87
14.2 Learning as Optimisation 87
14.3 Explanation Based Learning/ Generalisation 88
14.4 Operationality 88
14.5 Definition of EBL 88
    14.5.1 Inputs 88
    14.5.2 Output 88
14.6 A Logic Interpretation 88
    14.6.1 Explanation 88
    14.6.2 Generalisation 89
    14.6.3 Result 89
14.7 The generalisation process (Regression) 89
14.8 Prolog Code for EBL 89
14.9 EBG = Partial Evaluation 90
14.10Reading 90

15 Examples of EBL in Practice 91
15.1 STRIPS MACROPS 91
15.2 Evaluation of EBL 93
15.3 LEX2 - Learning Symbolic Integration 93
15.4 SOAR - A General Architecture for Intelligent Problem Solving . 95
15.5 Using EBL to Improve a Parser 96
15.6 References 96

16 Unsupervised Learning **97**
16.1 Mathematical approaches to Unsupervised Learning 97
16.2 Clustering 97
16.3 Principal components analysis 99
16.4 Problems with conventional clustering 100
16.5 Conceptual Clustering 100
16.6 UNIMEM 100
16.7 COBWEB 101
16.7.1 Category Utility 101
16.7.2 The Algorithm 102
16.7.3 Comments on COBWEB 102
16.8 Unsupervised Learning and Information 102
16.9 References 103

17 Knowledge Rich Learning - AM **105**
17.1 Mathematical Discovery as Search 105
17.2 The Architecture of AM 105
17.2.1 Representation of Concepts 105
17.2.2 The Agenda 106
17.2.3 The Heuristics 107
17.3 Types of Knowledge given to AM 108
17.4 Performance of AM 108
17.5 Conclusions 109
17.6 Reading 109

18 Theoretical Perspectives on Learning **111**
18.1 Gold - Identifiability in the Limit 111
18.2 Valiant - PAC Learning 112
18.3 Criticisms of PAC Learning 113
18.4 Reading 114

A Appendices **115**
A.1 Principal Components and Eigenvectors 115

Chapter 1

Machine Learning - Overview

This chapter attempts to give a taste of the kinds of things that Machine Learning involves. Unfortunately there are many references to topics that will be covered in more detail later. The reader is advised to read this chapter again after seeing all the material.

1.1 A Definition

Simon gives the following definition of learning:

> "Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time"

Learning involves doing better at *new* tasks that have not been previously encountered - therefore learning much involve some kind of *generalisation* from past experience. Simon's definition allows for many different kinds of learning systems. In particular, a system that is able to reorganise or reformulate its knowledge into a more compact or useful form could be said to be learning (cf FINGER and the work on EBG that we will see in Chapter 14).

1.2 Some Overall Comments

- There is a vicious circle connected with learning and performance. It is pointless trying to build a learning system until you know what kind of representations will actually be useful in performance. On the other hand, it is in some sense pointless developing clever representations for performance that cannot be learned. The only solution seems to be to pursue both lines of research in parallel.

- In 1978, PhD students in DAI were advised "Don't do anything on learning, if you can fight the urge". At that time, there were relatively few standard techniques in the area and it was very difficult to find research topics that were not hopelessly ambitious. Since then there have, however, been a number of well-defined and useful techniques developed from which new research has blossomed. This has been reflected in the number of papers on machine learning presented at international AI conferences like IJCAI. In 1991, for the first time, machine learning was the area with the largest number of papers submitted to IJCAI.

- Is machine learning really a proper subfield of AI? When we look at learning systems in different areas, we will see many differences between them. A general theory of machine learning that can encompass all of these has yet to emerge. So some people doubt whether machine learning is a coherent discipline in its own right. I think that it is too early to give a good answer to this question. But it is interesting that there are similarities across different learning systems that become apparent when one looks a bit below the surface.

1.3 Parameters of a Learning System

In this section we will attempt to list some of the main parameters that affect the design of a machine learning system.

1.3.1 Domain/ Knowledge Representation

Clearly a learning system depends greatly on the knowledge representation scheme in which its outputs are expressed and the extent to which knowledge is available to the learning process. Some learning approaches are quite "knowledge free" (such as the simpler distance-based approaches of Chapter 7); others, such as AM (Chapter 17) are more "knowledge intensive".

1.3.2 Application type

From Simon's definition, it follows that learning systems can be expected to produce all sorts of different kinds of outputs. Here are some of the more common tasks that machine learning is asked to address:

Concept Learning. The aim is, given examples (and usually also non-examples) of a given concept, to build a representation from which one can judge whether new observations are examples or not.

Classification. This is the generalisation of concept learning that is obtained when there are multiple concepts and the task is to build a system that can determine which of them fits a new piece of data best.

Rule Induction. This can be regarded as the subcase of classification learning when the output is to be expressed in terms of 'if-then' rules. The term "rule induction" can also cover systems that develop more complex rules (for instance, logic programs) from specific data. A related task is refining an existing set of rules so that they better fit some particular data.

1.3.3 Type of Input

The Handbook of Artificial Intelligence distinguishes between the following kinds of learning. Here what is at issue is mainly the kind of information that the learner receives.

- Rote learning. For instance, Samuel wrote a program for playing checkers which used the principle of minimax search. Whenever the system calculated the evaluation of a board position by looking ahead, the system remembered that position and the backed-up value. This meant that next time the system would achieve extra look ahead for free by using the stored evaluation, rather than repeating the calculation (and possibly stopping prematurely because of hitting the search depth bound).

- Learning by being told. This seems to be associated with the idea of a program that takes "advice" in a very neutral form and "operationalises" it, i.e. converts the advice into a form that can be directly used in the program. There do not seem to be many programs of this kind.

- Learning from examples (supervised learning). Most of the learning systems that we will look at fall into this category.

- Learning by analogy. Here the task is to transform knowledge from one domain into useful knowledge in another. This is a very difficult problem, and one that we will not consider further.

- Learning by doing. This seems to cover systems that automatically optimise their representations as they perform in the world. This might, thus cover Samuel's program and the reinforcement learning systems of Chapter 12.

- Learning by observation (unsupervised learning). Here the goal of learning is to detect interesting patterns in the environment. This involves, for instance, detecting similarities between different observations.

1.3.4 Types of Interaction

The last set of distinctions seem to assume that it is really the human teacher who is in control of the learning process. In practice, the learning system can sometimes be more efficient if it is allowed to ask questions to the teacher/ world. For instance, a concept learning system could hypothesise its own examples and ask the teacher whether or not they were instances of the concept. This is, however, not always possible as there may only be a fixed set of training data available and being available to answer a learning system's questions might require significant human time investment. *Discovery systems* attempt to learn by designing and carrying out experiments, in the same way that a scientist might develop new theories.

If a human teacher is involved with a learning system, there are still choices as to the timing of the interaction. In *incremental* learning, the system maintains at each point (after each input is received) a representation of what it has learned (and can in principle answer questions from this representation). Sometimes incremental learning will not be efficient or possible, in which case the system will collect a number of inputs and process them in "batch mode".

1.4 Some Views of Learning

Here we present some particular views of learning (not necessarily disjoint).

1.4.1 Learning as Reinforcement

In this view, the operation of the learning system goes through the following cycle:

1. Training
2. Evaluation
3. Credit/Blame Assignment
4. Transformation

Thus the system receives examples from a trainer. On the basis of evaluating its performance, it decides which of its representations are performing well and which badly. On the basis of this, it performs some transformations on them. It then gets another example, and so on. Minsky highlights the "basic credit assignment problem for complex reinforcement learning systems" as particularly difficult - in general, how is a complex system to know which of its parts are particularly responsible for a success or a failure?

The noughts and crosses program is an example of reinforcement learning. So are approaches based on Connectionism and Genetic Algorithms (which are treated more fully in other modules).

1.4.2 Learning as Search

In this view, there is a space of possible concepts that could be learned. The task of the learner is to navigate through this space to find the right answer. This view is the basis of the version spaces method for concept learning (Mitchell) and also of the operation of discovery systems like AM.

When learning is viewed as search, different search methods will give rise to different kinds of learners. In this module, we will see how some new search strategies (gradient descent and genetic mutation) yield particular kinds of learners.

1.4.3 Learning as Optimisation

In this view, learning is seen as the process of transforming knowledge structures into new ones that are more useful/ efficient/ compact. "Chunking" is one example of this. Generalisation is also a way of producing a compact representation of bulky data.

The FINGER program can be thought of as a simple example of optimisation learning. The work that we will see on EBG (Explanation Based Generalisation) is also in this spirit. We will also see learning viewed as a process of minimising entropy (ID3) and maximising category utility (COBWEB). In all these cases, it is more a *reformulation* of existing knowledge that is carried out, rather than "learning in a vacuum".

1.4.4 Learning as Curve Fitting

In this view, learning is seen as the task of finding optimal values for a set of numerical parameters. Our simple "function learning" program was an example of this. A more serious example is Langley's BACON system, which attempts to make sense of numerical data in Physics and Chemistry:

Time (T)	Distance (D)	D/T	D/T^2
0.1	0.098	0.98	9.8
0.2	0.390	1.95	9.75
0.3	0.880	2.93	9.78
0.4	1.572	3.93	9.83
0.5	2.450	4.90	9.80
0.6	3.534	5.89	9.82

BACON must actually hypothesise the *form* of the equation linking the different variables, as well as find any numerical parameters in the equation. In fact, it makes no sense to ask a learning system to find "the" curve that goes through a given set of points, because there are infinitely many such curves! It is necessary to specify what kinds of curves we prefer. Learning is only possible if we give it an appropriate *bias*.

Connectionist learning systems are an excellent example of the "learning as curve fitting" view.

1.5 Applications of Machine Learning

Machine learning systems have led to practical applications that have saved companies millions of pounds (see the Langley and Simon paper cited below for more details). Applications have included helping banks to make credit decisions, diagnosing and preventing problems with mechanical devices and automatically classifying celestial objects.

There is a great deal of industrial interest in the area of *data mining*, which combines together methods from databases, statistics and machine learning to help companies find useful patterns in data that they have collected. The aim of this enterprise is to summarise important knowledge that may be implicit in unanalysed data and which can be put to useful work once it has been discovered. Data mining probably involves more unsupervised than supervised learning, though once one has formulated an idea as to which information one would like to predict on the basis of which other information then supervised learning immediately becomes relevant.

1.6 The ML Module

There is no really good way to present the whole of Machine Learning in a linear sequence, and I am constantly thinking of new ways to attempt this. The rest of this module will be split into the following sections (though these topics are not disjoint):

1. Concept learning (chapters 2 to 5).
2. Classification learning - numerical approaches (chapters 6 to 8).
3. Classification learning - learning discrimination trees (chapters 9 to 11).
4. Reinforcement learning (chapters 12 and 13).
5. Learning for optimisation (explanation based learning) (chapters 14 and 15).
6. Unsupervised learning (chapters 16 and 17).
7. Theoretical approaches and conclusions (chapter 18).

1.7 Reading

See Thornton Chapter 2 for an alternative presentation of many of the ideas here.

- Langley, P., Bradshaw, G. and Simon, H., "Rediscovering Chemistry with the BACON system", in Michalski, R., Carbonell, J. and Mitchell, T., Eds., *Machine Learning: An Artificial Intelligence Approach*, Springer Verlag, 1983.

- Simon, H. A., "Why Should Machines Learn?", in Michalski, R. S., Carbonell, J. G. and Mitchall, T. M., *Machine Learning: An Artificial Intelligence Approach*, Springer Verlag, 1983.

- Cohen, P. R. and Feigenbaum, E. A., *Handbook of Artificial Intelligence* Vol 3, Pitman, 1982.

- Langley, P. and Simon, H., "Applications of Machine Learning and Rule Induction", *Communications of the ACM* Vol 38, pp 55-64, 1995.

- Minsky, M., "Steps Toward Artificial Intelligence" p432-3 in Feigenbaum, E. and Feldman, J., Eds., *Computers and Thought*, McGraw Hill, 1963.

Chapter 2

Concept Learning - Description Spaces

Concept learning is the task of coming up with a suitable description for a concept that is consistent with a set of positive and negative examples of the concept which have been provided.

2.1 Types of Observations

In general, a learning system is trying to make sense of a set of *observations* (a *sample* from the total population of possible observations). In concept learning, the observations are the instances that are supplied as positive and negative examples (and possibly also as data for testing the learned concept). We can usually characterise an observation in terms of the values of a fixed number of *variables* (or *features*), which are given values for all observations.

To be precise, imagine that we have m different variables that are measured for each observation. These might measure things like size, importance, bendiness, etc. For each observation, each variable has a value. In mathematical models, these values will be numerical, but in general they could also be symbolic. Let us use the following notation:

x_{ij} is the ith measurement of variable x_j

If there are n observations in the training sample, then i will vary from 1 to n. j varies from 1 to m (the number of variables). The ith observation can then be represented by the following vector:

$$\begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{im} \end{pmatrix}$$

This can be viewed as the coordinates of a point in m-dimensional space. This idea is most natural when the values are all continuously-varying numbers, but can be extended if we allow the "axes" in our m-dimensional graph also to be labelled with symbolic values (where often the relative order does not matter).

Thus the training examples of a learning system can be thought of as a set of points in m-dimensional space. A concept or class of objects also corresponds to a set of points in this space. The job of a concept learner is to generalise from a small set of points to a larger one.

2.2 Types of Descriptions/Concepts

The result of concept learning is a description (concept) which covers a set of observations. Mathematically, a concept is simply a function that determines, for a new observation, whether that observation lies in the class or not. That is, a function g such that:

$g(\mathbf{x}) > 0$ if x is an instance of the concept
$g(\mathbf{x}) \leq 0$ otherwise

Such a function is called a *discriminant function.*

In general, although for purely numerical data it may suffice to have g implement some obscure numerical calculation, if the data is symbolic or if we wish to integrate the result of learning with other systems (e.g. expert systems) we will want to restrict the form of this function so that it corresponds to a human-readable description of those individuals belonging to the class (e.g. something in terms of some kind of logic). We need to consider what such descriptions might look like. In deciding on a language for expressing concepts, we are expressing a first kind of *bias* for our learning system (*representational bias* - the other kind of bias, *search bias* will be introduce a strategy for searching for the right concept, which inevitably will result in some possibilities being considered before others). The bias will limit the possible concepts that can be learned in a fundamental way. The choice of an appropriate description language is, of course, an important way of building knowledge into the learning system.

Although we might need other kinds of descriptions as well, the least we want to have available is *conjunctive* descriptions which say something about each feature (in the same way that observations do). Such descriptions correspond to hypercubes in the m-dimensional space, given an appropriate ordering of symbolic values. The new descriptions that we will introduce will differ from observations by being able to say less specific (more general) things about the values of the features. What possible generalisations can be stated depends on the nature of the feature and its possible values, with the following being the most usual possibilities:

Nominal values. There is a finite set of possible values and there are no important relationships between the different values, apart from the fact that it only makes sense to have one value. For instance, the nationality of a person must be one of the current countries in the world. In this case, the only generalisation that can be stated is "anything" (i.e. no restriction is placed on the value of this feature).

Ordered values. This case is similar in many ways to the last one, except that it makes sense to compare the values. For instance, the number of children of a person must be a whole number, and so one can talk in terms of "having at least 2 children" say. For ordered values, as well as generalising to "anything" we can also generalise to any *range* of possible values.

Structured values. . This is for the case where possible symbolic values can be related to one another by the notion of abstraction. For instance, an observation of a meal could be described in terms of its main component, but some possible values could be more general than others. If the main component of a meal is pork then it is also meat. In this case, a space of possible generalisations for the feature is given in advance.

2.3 Abstract Characterisation of Description Spaces

Although the above examples consider the most frequent ways that observations and descriptions are characterised in Machine Learning, we can describe many concept learning algorithms in a way which abstracts from the particular details of how the descriptions are made up. The key idea that we need is that of a *description space* D, which is a set of possible descriptions related by *subsumption*:

d_1 is subsumed by d_2 ($d_1 \leq d_2$) if
d_1 is a less general description than d_2

In terms of the $\leq$ relation, a description of an individual (e.g. an example or non-example of a concept) is a minimal description (i.e. is one of the least general descriptions). Sometimes it is useful to think of an even less general description $\perp$ ("bottom"), which is so specific that it cannot apply to any individual. In the other direction, there are more general descriptions which can apply to many individuals. At the extreme, there is the description $\top$ ("top") which is so general that it applies to all individuals.

In terms of description spaces, concept learning can be characterised as follows. We are trying to learn a concept d (the "target") which is somewhere in the description space and the only information we are given about d is of the following kinds:

- $d_1 \leq d$ (d_1 is an *example* of d)

- $d_1 \not\leq d$ (d_1 is *not an example* of d)

for some descriptions d_1.

In practice, in order to organise searches around a description space, it is useful to be able to get from a description d to those descriptions $\nearrow d$ which are "immediately more general" than it and to those descriptions $\searrow d$ which are "immediately more specific" than it. One says that descriptions in $\nearrow d$ "cover" d and that d "covers" the descriptions in $\searrow d$. Description spaces are often displayed diagrammatically, with each description placed immediately below, and linked with a line to, the descriptions that cover it (hence the direction of the arrows in our notation). $\nearrow$ and $\searrow$ are defined formally as follows:

$$\nearrow (d) = \text{MIN } \{d_1 \epsilon D \mid d \leq d_1\}$$
$$\searrow (d) = \text{MAX } \{d_1 \epsilon D \mid d_1 \leq d\}$$

[1] To get from a description d to the more general versions in $\nearrow d$ is generally achieved by applying *generalisation operators* to d. For instance, a generalisation operator might remove one requirement from a conjunctive description. Similarly to get from a description d to the more specific versions in $\searrow d$ is generally achieved by applying *refinement operators* to d. For instance, a refinement operator might add one extra requirement to a conjunctive description.

```
discriminate($d_1$,$d_2$):
    $Res := \{\}$      ;;; set of results
    $Set := \{d_1\}$  ;;; descriptions to refine
    until $Set = \{\}$ do
        $Set := \bigcup_{s \epsilon Set}$ nextd($s, d_2$)
    return MAX $Res$

nextd($d_1$,$d_2$):
    if ($d_2 \not\leq d_1$) then
        $Res := Res \cup \{d_1\}$
        return {}
    else
        return $\searrow d_1$)
```

Figure 2.1: Algorithm for 'discriminate'

A candidate concept to explain a given set of data, d_1 say, will have to be changed if it fails to account for a new piece of data. If it fails to account for a

---

[1] MAX and MIN select just the maximal/minimal elements from a set:

$$\text{MAX } F = \{x \epsilon F \mid \text{for all } y \epsilon F, \text{ if } x \leq y \text{ then } x = y\}$$
$$\text{MIN } F = \{x \epsilon F \mid \text{for all } y \epsilon F, \text{ if } y \leq x \text{ then } x = y\}$$

positive example d_2, then in general we wish to make it minimally more general such that it does indeed subsume that example. The function 'generalise' gives us a set of possibilities for how to do this:

$$generalise(d_1, d_2) = \text{MIN } \{d \epsilon D \mid d_1 \leq d, d_2 \leq d\}$$

Similarly, if d_1 fails to account for a new negative example d_2, it will need to be made more specific so that it no longer subsumes d_2. The function 'discriminate' gives us the possibilities for this:

$$discriminate(d_1, d_2) = \text{MAX } \{d \epsilon D \mid d \leq d_1, d_2 \not\leq d\}$$

One possible algorithm to compute 'discriminate' is shown in Figure 2.1. Note that it uses the 'covers' function $\searrow$ to "move down" incrementally through the description space until it finds descriptions that no longer subsume d_2. Similarly the computation of 'generalise' may well involve using $\nearrow$.

2.4 Examples of Description Spaces

Let us now consider how these abstract operations would work with the concrete types of descriptions we have considered above.

2.4.1 Nominal Features

An example of a description space arising from a single nominal feature ("colour") is shown in Figure 2.2. Where a description just refers to one nominal feature, the only possible cases for $x \leq y$ are where x and y are the same, x is $\perp$ ("nothing") or y is $\top$ ("anything"). $\nearrow$ goes up from normal values to $\top$ and from $\perp$ to all possible normal values ($\searrow$ goes the other way). 'discriminate' only produces

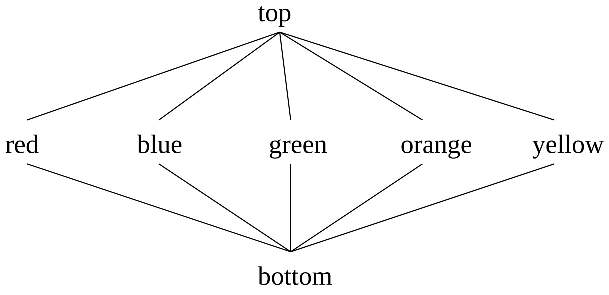


Figure 2.2: Description space for a nominal feature

interesting results when used with $\top$ and 'generalise' produces $\top$ unless one of the arguments is $\bot$. For instance,

$$\begin{aligned}
\nearrow blue &= top \\
\searrow top &= \{red, blue, green, orange, yellow\} \\
generalise(red, blue) &= \{top\} \\
discriminate(top, red) &= \{blue, green, orange, yellow\}
\end{aligned}$$

2.4.2 Features with ordered values

An example description space arising from a single feature ("number of legs", perhaps) with ordered values is shown in Figure 2.3. Here the basic values are the integers 1 to 4. Generalisation introduces the possible ranges, so that for instance [1,3] indicates the concept of something with between 1 and 3 legs. For compatibility, the simple integer values x are shown in the format [x,x]. In such a description space, one range subsumes another if it includes it. Thus [1,3] subsumes [2,3]. $\nearrow$ gives the ranges at the next "level" which include a given range, and $\searrow$ does the reverse. 'generalise' produces the union of two intervals and 'discriminate' produces the largest subintervals of the first argument that do not include the second argument. For instance,

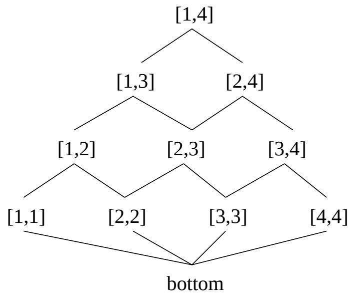


Figure 2.3: Description space for a feature with ordered values

$$\begin{aligned}
\nearrow [2,3] &= \{[1,3],[2,4]\} \\
\searrow [2,3] &= \{[2,2],[3,3]\} \\
generalise([1,3],[2,4]) &= \{[1,4]\} \\
discriminate([1,4],[2,3]) &= \{[1,2],[3,4]\}
\end{aligned}$$

2.4.3 Structured Features

Figure 2.4 shows an example of a description space arising from one structured feature ("vehicle type"). Here subsumption (and all the other operations) is determined, not by any general principles but simply by whatever structure is specially described for the values of the feature. For instance:

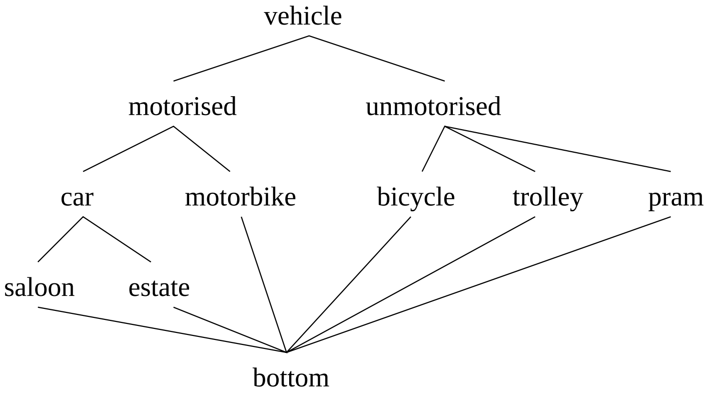


Figure 2.4: Description space for a structured feature

$$\begin{aligned}
\nearrow saloon &= \{car\} \\
\searrow car &= \{saloon, estate\} \\
generalise(saloon, bicycle) &= \{vehicle\} \\
discriminate(vehicle, bicycle) &= \{motorised, trolley, pram\}
\end{aligned}$$

2.4.4 Bundles of Independent Features

Where descriptions consist of values for independent features, one description subsumes another just in case the corresponding values subsume one another for each feature. How that works depends on the types of features (as discussed above). Figure 2.5 shows the description space arising from having two features, the first being a "colour" feature (as above, but with red and blue as the only values) and the second being a "vehicle type" feature (as above, but confined to the motorised part of the taxonomy and missing out the possible value "motorised"). Computing $\nearrow$ just involves using $\nearrow$ on one of the feature values (leaving the others unchanged), and similarly for $\searrow$ and 'discriminate'. Put another way, the refinement operators are simply the refinement operators for the individual features, and similarly for the generalisation operators. However 'generalise' involves generalising the values for *all* the features. For instance, in this example:

$$\nearrow redcar = \{redvehicle, topcar\}$$

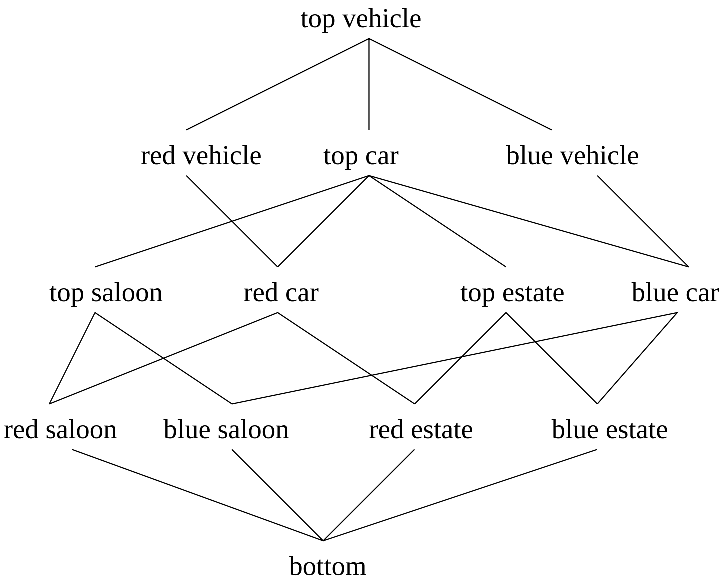

top vehicle
red vehicle
top car
blue vehicle
top saloon
red car
top estate
blue car
red saloon
blue saloon
red estate
blue estate
bottom

Figure 2.5: Description space for two features

$$\begin{aligned} \searrow redcar &= \{redsaloon, redestate\} \\ generalise(bluesaloon, blueestate) &= \{bluecar\} \\ discriminate(topvehicle, redestate) &= \{bluevehicle, topsaloon\} \end{aligned}$$

Here is how it looks in general for descriptions taking the form <v1,v2>, where v1 is a value for feature1 and v2 is a value for feature2 (the situation for m features follows a similar pattern). Don't worry about the mathematics here, as long as you understand the first two paragraphs of this subsection.

$$\begin{aligned} < a_1, a_2 > \leq < b_1, b_2 > \quad iff &\quad a_1 \leq b_1 \ and \ a_2 \leq b_2 \\ \nearrow < a_1, a_2 > \quad = &\quad \bigcup_{g \epsilon \nearrow a_1} \{< g, a_2 >\} \cup \bigcup_{g \epsilon \nearrow a_2} \{< a_1, g >\} \\ \searrow < a_1, a_2 > \quad = &\quad \bigcup_{g \epsilon \searrow a_1} \{< g, a_2 >\} \cup \bigcup_{g \epsilon \searrow a_2} \{< a_1, g >\} \\ generalise(< a_1, a_2 >, < b_1, b_2 >) \quad = &\quad \bigcup_{a \epsilon generalise(a_1, b_1), b \epsilon generalise(a_2, b_2)} \{< a, b >\} \\ discriminate(< a_1, a_2 >, < b_1, b_2 >) \quad = &\quad \bigcup_{d \epsilon discriminate(a_1, b_1)} \{< d, a_2 >\} \cup \\ &\quad \bigcup_{d \epsilon discriminate(a_2, b_2)} \{< a_1, d >\} \end{aligned}$$

Chapter 3

Concept Learning - Search Algorithms

3.1 Search Strategies for Concept Learning

The task of concept learning is a kind of search task - looking for a concept in the relevant description space which correctly accounts for both the positive and the negative observations. There are a number of different strategies that can be used:

Incremental search reads in the positive and negative examples one by one, moving around the search space as it does so. On the other hand, nonincremental search takes into account all of the examples at once. Here we will concentrate on incremental algorithms.

General-Specific search starts with very general concepts and searches "downwards" as required by the data. On the other hand, specific-general search starts at the "bottom" and goes to more general concepts only as required by the data.

Exhaustive search covers the whole search space and so is guaranteed to find all solutions. On the other hand, *heuristic search* attempts to limit the number of possibilities considered, at the possible expense of missing the (best) solution. Here we will concentrate almost entirely on exhaustive strategies.

An algorithm for incremental general-specific search is shown in Figure 3.1. This algorithm stores all the positive examples encountered in a set $PSET$ (though it does not need to store the negative examples). It maintains in the set G all the *most general* concepts which are consistent with the data seen so far. The elements of G have to be made more specific (by 'discriminate') to deal with the negative examples, and at all points the elements of G have to be more

```
$PSET$ := {} ;;; stored positive examples
$G := \{\top\}$ ;;; current set of solutions
repeat until no more data
    read next data item $d$
    if $d$ is a positive example,
        $PSET := PSET \cup \{d\}$
        $G := \{g \epsilon G \mid d \leq g\}$
    else if $d$ is a negative example,
        $G :=$ MAX $\bigcup_{g \epsilon G}$ discriminate$(g, d)$
        $G := \{g \epsilon G \mid \forall d \epsilon PSET \; d \leq g\}$
return $G$
```

Figure 3.1: Incremental General-Specific Search

```
$NSET$ := {} ;;; stored negative examples
$S := \{\bot\}$ ;;; current set of solutions
repeat until no more data
    read next data item $d$
    if $d$ is a negative example,
        $NSET := NSET \cup \{d\}$
        $S := \{s \epsilon S | d \not\leq s\}$
    else if $d$ is a positive example,
        $S :=$ MIN $\bigcup_{s \epsilon S}$ generalise$(s, d)$
        $S := \{s \epsilon S \mid \forall d \epsilon NSET \; d \not\leq s\}$
    endif
return $S$
```

Figure 3.2: Incremental Specific-General Search

general than all the positive examples (this needs to be retested when a new positive example arrives and also when elements of G are made more specific). The elements of G only get more specific, so once we have ensured that they don't subsume a given negative example, we never need to check that again. A very similar algorithm for incremental specific-general search is shown in Figure 3.2. In incremental general-specific search, we use 'generalise' to make the specific descriptions more general to cover positive examples. We need to store all encountered negative examples and retest against them whenever we make elements of the S set more general.

3.2 Version Spaces

Mitchell devised a bidirectional concept learning search algorithm which combines together the best points of the two above algorithms. The result, has the following properties:

- It is an exhaustive search algorithm which makes no unwarranted commitments (and hence excludes no possible solutions).
- It is seriously incremental, in that it does not need to store any of the positive or negative examples.

Whereas each of the previous algorithms only maintained a single solution set, either G (most general consistent descriptions) or S (most specific consistent descriptions), the CEI maintains both. The combination of a G and an S set provides a way of representing the complete set of possibilities consistent with the data seen so far - the *version space*. The set of possibilities for the target concept allowed by G and S, $VS < G, S >$ in our notation is in fact as follows:

$$VS < G, S > = \\ \{d \epsilon D \mid \text{for some } s \epsilon S,\ s \leq d \text{ and for some } g \epsilon G,\ d \leq g\}$$

Three particular important special cases can arise:

1. If G is $\{\top\}$ and S is $\{\bot\}$ then the set represented is the whole of the description space. This is how a learning system will start off.
2. If G and S both become empty, then the set represented is also empty (there are no concepts d that satisfy all the requirements we have set).
3. If G is $\{x\}$, S is $\{y\}$, $x \leq y$ and $y \leq x$ then the set represented is also $\{x\}$ (or $\{y\}$, which is equivalent to it) and the target concept must be x.

3.3 The Candidate Elimination Algorithm

Mitchell's algorithm, the "candidate elimination algorithm" is best thought of as having a number of components.

3.3.1 Pruning a Version Space Representation

The following operations can be used to remove unnecessary elements from a version space representation $VS < G, S >$:

1. If $s \epsilon S$ and for all $g \epsilon G$, $s \not\leq g$, then s can be removed from S
2. If $g \epsilon G$ and for all $s \epsilon S$, $s \not\leq g$, then g can be removed from G

3. If distinct s_1 and $s_2 \epsilon S$ and $s_1 \leq s_2$, then s_2 can be removed from S
4. If distinct g_1 and $g_2 \epsilon G$ and $g_1 \leq g_2$, then g_1 can be removed from G

3.3.2 Applying a Version Space Representation

Even before concept learning has converged to a single target, a version space representation $VS < G, S >$ can be used to determine whether a new description d_1 must be or must not be an example of the target d, as follows:

- If ($\forall$ $g \epsilon G$, $d_1 \not\leq g$) then $d_1 \not\leq d$ (d_1 is definitely not an example)
- Otherwise if ($\forall$ $s \epsilon S$, $d_1 \leq s$) then $d_1 \leq d$ (d_1 is definitely an example)
- Otherwise it is not known whether $d_1 \leq d$ (whether d_1 is an example or not)

3.3.3 Dealing with a Positive Example

When a new piece of data d_1 is discovered which is an example of the target concept d ($d_1 \leq d$), the two sets are updated as follows:

- $G := G$
- $S := \bigcup_{s \epsilon S} \ generalise(s, d_1)$

3.3.4 Dealing with a Negative Example

When a new piece of data d_1 is discovered which *not* an example of the target concept d ($d_1 \not\leq d$), the two sets are updated as follows:

- $G := \bigcup_{g \epsilon G} \ discriminate(g, d_1)$
- $S := S$

3.3.5 The Algorithm

The complete algorithm can now be stated in Figure 3.3.

3.4 Disjunctive Descriptions

With all these search methods, the system can only learn those descriptions in the description space. As we have seen, this is described as the learning system having *bias*.

```
$G := \{\top\}$
$S := \{\bot\}$
until there is no more data
   read the next observation $d_1$
   if $d_1$ is meant as a question,
      answer it according to section 3.3.2
   otherwise if $d_1$ is meant as a new positive example,
      update $S$ and $G$ according to section 3.3.3
      prune $S$ and $G$ according to section 3.3.1
   otherwise if $d_1$ is meant as a new negative example,
      update $S$ and $G$ according to section 3.3.4
      prune $S$ and $G$ according to section 3.3.1
   if the version space is now empty or just contains a
    single element (section 3.2), exit
```

Figure 3.3: The Candidate Elimination Algorithm

If we want to learn disjunctive and negative descriptions, then it is natural to want to have $a \vee b$ and $\neg a$ in the description space, for all descriptions a and b. But if this is the case, then after a sequence of data including the examples $e_1, e_2, \ldots e_m$ and the non-examples $ne_1, ne_2, \ldots ne_n$, the version space will be:

$$\text{G} = \{\neg ne_1 \vee \neg ne_2 \vee \ldots \neg ne_m\}$$
$$\text{S} = \{e_1 \vee e_2 \vee \ldots e_n\}$$

This is correct, but uninteresting. Essentially the system has learned the data by rote and has not been forced to make any interesting generalisations. Having arbitrary disjunctions and negations in the description space has effectively removed all bias – it can learn anything whatsoever. Bias of some form is essential if a learning system is to do anything interesting (e.g. make inductive “leaps”).

3.4.1 AQ

Michalski’s AQ algorithm is one “solution” to the problem of learning disjunctive concepts. Although AQ was originally described in a rather different way, we will present it as if it used version spaces. The algorithm learns a description of the form $g_1 \vee g_2 \vee \ldots g_n$. Essentially it uses Candidate Elimination (with a description space not allowing disjunctions) to learn each of the g_i, according to the following algorithm:

1. Pick a positive example e_i.
2. Set $G = \{\top\}$, $S = \{e_i\}$.

3. Update this version space for all the negative examples in the data. Each description in G now covers e_i but no negative examples.

4. Choose some element of G to be the next g_i.

5. Remove all examples of g_i from the data.

6. Repeat 1-5 until there are no positive examples left in the data.

NB The selection of the e_i and g_i is obviously very important. The resulting disjunctive description is more like the "G" of a concept than its "S" (i.e. it is a very general concept that excludes all the negative examples, rather than a very specific concept that includes all the examples). This is basically a "greedy" hill-climbing algorithm.

3.5 Reading

Chapter 2 of Langley has descriptions of a number of different concept learning search strategies. The Candidate Elimination Algorithm was originally developed in Mitchell's PhD thesis. There is a reasonable description of it in Chapter 2 of Thornton and in Volume 3 of the Encyclopaedia of AI.

Thornton Chapter 4 is a description of AQ, but it is rather misleading in some ways. The Encyclopaedia of AI (Volume 3, pp423-428) has some brief material on AQ. An article by Mitchell in *Readings in Machine Learning* argues that bias is essential in a learning system.

- Langley, P., *Elements of Machine Learning*, Morgan Kaufmann, 1996.

- Mitchell, T. M., "Generalisation as Search", *Artificial Intelligence* Vol 18, 1982 (Also in *Readings in Machine Learning*).

Chapter 4

Inductive Logic Programming 1

Inductive Logic Programming is the research area that attempts to build systems that build logical theories from examples. ILP is a kind of concept learning, but where the language for expressing concepts (logic) is more powerful than the simple examples we have seen in the last lectures. In many ways, the term ILP is misleading as, although most researchers restrict themselves to the Prolog subset of logic, the techniques developed are ways of building systems of logic axioms, independently of whether these are the be viewed as logic programs or not.

In this lecture, we describe Shapiro's MIS system, probably the first significant work in the ILP area. ILP is now a flourishing subarea of machine learning in its own right, and we describe some more recent work in the next lecture.

4.1 The Problem

Imagine that a learning program is to be taught the Prolog program for "quick sort", given a definition of the predicates `partition`, `append`, `=<` and `>`. It is provided with a set of positive ground examples, such as `qsort([1,0],[0,1])`, and negative examples, such as `qsort([1,0],[1,0])`. We might hope that such a program would eventually come up with the following:

```
qsort([],[]).
qsort([H|T],Result) :-
   partition(H,T,List1,List2),
   qsort(List1,Res1),
   qsort(List2,Res2),
   append(Res1,[H|Res2],Res).
```

In general, given the following:

1. Some background knowledge $\mathcal{K}$ (here, the definitions of `append`, etc.).
2. A set of positive examples ϵ^+ (here, ground examples of `qsort`), such that $\mathcal{K} \not\vdash \epsilon^+$.

3. A set of negative examples ϵ^-.

the task is to find a set of clauses $\mathcal{H}$ such that:

1. $\mathcal{K} \wedge \mathcal{H} \vdash \epsilon^+$
2. $\mathcal{K} \wedge \mathcal{H} \nvdash \epsilon^-$

Thus we can see that inductive logic programming is a kind of concept learning, where the description of the concept is constrained to be a set of clauses $\mathcal{H}$. If we imagine that an ILP program can learn a Prolog program like:

```
p(X) :- q, r, s.
p(X) :- t, u, v.
```

then it has essentially learned a description of the concept `p` rather like:

$$(q \wedge r \wedge s) \vee (t \wedge u \wedge v)$$

i.e. a disjunction of conjunctions, which is exactly the kind of representation that a system like AQ builds. Thus, although learning logic descriptions looks rather different from learning simple feature-value descriptions, in fact we may expect to see some similarities with the concept learning approaches we have already seen.

4.2 Architecture of MIS

Shapiro's MIS (Model Inference System) has a structure similar to that of a classic reinforcement learning system:

1. Training
2. Evaluation
3. Credit/Blame Assignment
4. Transformation

At any point, the system has a set of clauses $\mathcal{H}$ that accounts for the examples and non-examples seen so far. It then takes a new example or non-example and changes the clauses appropriately. Note that the transformations undertaken involve things like adding a new clause orr changing and existing one. This is quite different from the numerical operations carried out in reinforcement learners like Michie's MENACE and connectionist systems. For this reason we might choose not to call MIS a reinforcement learning system.

A critical point in this learning process is determining which clause in the program to change (the credit assignment problem). In many ways, Shapiro's main contribution was to devise ways of locating the “bugs” that have to be fixed before a logic program can deal with a new example or non-example. Hence the title of his thesis “Algorithmic Program Debugging”. The basic algorithm used by MIS is as follows:

```
Set P to be the empty program
Let the set of known facts be empty
Let the set of marked clauses be empty
Until there are no more examples or non-examples,
  Read the next example or non-example to be dealt with
  Add it to the set of known facts
  Until P is correct on all known facts,
    If there is an example which P fails to prove,
      Find a goal A uncovered by P
      Search for an unmarked clause that covers A
      Add it to P
    If there is a non-example which P can prove,
      Detect a false clause
      Remove it from P and mark it
```

Clauses are marked if they are found to be incorrect - this is used to prevent the same clause being introduced into the program again later. The system also detects and corrects cases where the computation to prove a new fact does not terminate, though we will not consider that here.

4.3 New Positive Examples

If MIS receives a new example which it cannot prove, it has to add a new clause to the program being developed. Of course, it could simply add this new fact as an extra clause to the program. But this would not achieve any generalisation and would not integrate the new information with the existing theory. Adding a new fact really only makes sense if there is no existing clause that can be used (correctly) to reduce a proof of it to simpler goals. In such a case, we say that the fact is *uncovered* by the program. MIS reacts to a new positive example by searching for a fact (which may not be the example itself) in a failed proof of the example that is uncovered. It is not adequate to search for any subgoal of any attempted proof of the new example that fails - instead one must search for a goal *that should succeed* but fails. Hence the system needs to have information about which goals should succeed and with what solutions. This is called an *oracle*. In practice, MIS asks the user for this information. Note that the presence of an oracle means that MIS is different from a standard concept-learning program - it can ask questions as well as be given examples and non-examples.

The basic idea behind the algorithm for finding uncovered goals is given by the following Prolog program, where `ip(A,X)` means "If A is a finitely failing true goal then X is an uncovered true goal":

```
ip((A,B),X) :- A, !, ip(B,X).
ip((A,B),X) :- !, ip(A,X).
ip(A,X) :- clause(A,B), satisfiable(B), !, ip(B,X).
ip(A,A).
```

where the call `satisfiable(B)` asks the oracle whether the goal `B` has a solution and, if so, instantiates that goal to a true solution (it does not matter which one).

4.4 Refinement Operators

Once an uncovered true goal A has been found, it is necessary to find a clause that will cover it and add this to the program (notice that testing whether a clause covers a goal involves calling the oracle to see whether the body of the clause applied to the goal should be satisfiable or not). MIS basically enumerates possible clauses in a systematic way that allows whole branches of the search space to be pruned (because they cannot lead to any covering clause) as it goes along. It starts off with the most general possible clause and then enumerates new clauses in order of specificity. Once a clause has been found not to cover the goal, no more specific clause can possibly cover it. In terms of our previous approaches to concept learning, there is a description space of possible clauses and the system conducts an exhaustive general-specific search for a clause to subsume the goal.

The system goes from a clause to the set of "next most specific" versions of it (c.f. the $\searrow$ operator of the previous chapters) by using a set of *refinement operators*. Here are the operators used in one version of the system for creating a more specific clause q from a clause p:

- If p is empty, q can be the fact for any predicate with arguments which are distinct variables.
- q can be obtained by unifying together two variables in p.
- q can be obtained from p by instantiating a variable to a term with any functor and distinct new variables as arguments.
- q can be obtained from p by adding a goal to the body whose "size" is less than or equal to that of the head of the clause and where every variable occurring in the new goal already appears in the clause.

The main idea is conveyed by the following Prolog program, where the predicate `search_for_cover(P,C)` means "One of the possible clauses that covers goal `P` is `C`":

```
search_for_cover(P,C) :-
   functor(P,F,N),
   functor(P1,F,N),
   search_for_cover([(P1 :- true)],P,C).

search_for_cover([X|Rest],P,C) :-
   findall(Y,(refinement(X,Y),covers(Y,P)),List),
   check_refinements(List,Rest,P,C).

check_refinements(List,_,P,C) :-
   member(C,List), not_marked(C), !.
check_refinements(List,Rest,P,C) :-
   append(Rest,List,NewList),
   search_for_cover(NewList,P,C).
```

Basically, `search_for_cover/3` takes an "agenda" of clauses that cover `P` but which are inadequate since they are marked. This list starts off with the most general clause for the predicate of `P`. For each one, it enumerates (via `findall`) all the refinements of that clause that cover `P`. As soon as it gets to an unmarked clause it terminates, returning that clause. Otherwise, the new clauses get added to the end of the "agenda". This imposes a breadth-first search, producing increasingly specific clauses (though, of course, only the first solution is returned).

4.5 New Negative Examples

When a new negative example appears which the system can prove, it must look for a clause that is faulty and remove it. MIS does this by going through the "proof" of the non-example looking for a case where a clause is used in a way that is not valid (it has to consult the oracle for this).

The basic idea is shown in the following Prolog program, where `fp(A,X)` means "A is a solvable goal. If A is false then X is a false instance of a clause used in proving A; otherwise X=ok":

```
fp((A,B),X) :-
   fp(A,Xa), conjunct_fp(Xa,B,X).
fp(A,ok) :-
   system(A), !.
fp(A,X) :-
   clause(A,B), fp(B,Xb),
   clause_fp(Xb,A,(A:-B),X).

   conjunct_fp(ok,B,X) :- !, fp(B,X).
   conjunct_fp(X,_,X).
```

```
clause_fp(ok,A,_,ok) :- query_all(A), !.
clause_fp(ok,_,X,X) :- !.
clause_fp(X,_,_,X).
```

`conjunct_fp` is used here for processing the second of a conjunction of goals. If the first was "ok", then the fault must lie in the second; otherwise the answer is whatever fault was found in the first. `clause_fp` is used when we are considering a particular use of a clause. If there is a fault detected during the execution of the body, then that is returned as the answer (last clause). Otherwise the oracle must be consulted to see whether the conclusion of the clause really follows from this successful body. `query_all(A)` asks the oracle whether every instance of the argument `A` is true. If so, then there is no fault in this part of the proof tree; otherwise this is the faulty clause.

The actual algorithm used by MIS is slightly more complex than this. It analyses the sizes of the parts of the proof tree to determine which questions to ask the oracle first, in order to minimise the number of questions asked and the length of the computation.

4.6 Search

The above description of the MIS algorithm perhaps gives the impression that there is relatively little search involved. This would be misleading. When a covering clause for a true goal is sought, the first solution is chosen. If this is the wrong one, then this may not be discovered until some time later. Whenever a change is made to the program, the system has to check that all the known examples and non-examples remain properly accounted for. This may mean that the system has to keep coming back to the problem of finding a covering clause for some particular goal. Each time, it will search through the set of refinements from the start. Only the "memory" implemented via the marking of clauses will prevent it choosing the same clause again.

4.7 Performance and Conclusions

MIS is able to synthesise programs for predicates like `member`, `subsequence`, `subset`, `append` and `isomorphic` (for trees) from examples. It is able to improve on previous programs for learning LISP programs from examples. The weak point in MIS is the enumeration of possible clauses using the refinement operators. For simple examples, it will reach sensible clauses quickly but, since clauses are enumerated in order of complexity, for complex programs this search will be a serious problem.

4.8 Reading

- Shapiro, E., *Algorithmic Program Debugging*, MIT Press, 1982.

Chapter 5

Inductive Logic Programming 2

In this chapter we look at some more recent work on inductive logic programming.

5.1 Improving the Search - Quinlan's FOIL

Quinlan's FOIL system is an approach to inductive logic programming that adapts ideas from previous machine learning systems, namely:

- From AQ, a way of generalising from a set of examples that produces a disjunction of conjunctions.
- From ID3 (which we will discuss later), an information-theoretic heuristic to guide search.

In particular, the second of these allows FOIL to have a much more directed search through possible programs than does MIS.

5.1.1 Basic Characteristics

FOIL is given a set of examples and a set of non-examples for some predicate, as well as full definitions for subpredicates that it can use (it can also make recursive definitions). It constructs a set of clauses for the predicate. Each clause can only be of a restricted form - it can contain positive and negative goals in the body, but it cannot use function symbols (e.g. `[...]`) anywhere in the clause. This restriction is not as bad as it sounds, for instance, the recursive clause for `append` could be written:

```
append(X,Z,X1) :- list(X,Head,Tail), append(Tail,Z,Z1), list(X1,Head,Z1).
```

where

```
list([H|T],H,T).
```

is provided as a subpredicate.

Whereas MIS is (fairly) incremental, FOIL will only operate given the total set of examples and non-examples. Whereas MIS needs an oracle, FOIL operates without one.

5.1.2 Top-Level Algorithm

```
Set P to be the set of positive examples given
Until P is empty do
    Construct a clause that accounts for some elements of P and no
     negative examples.
    Remove the elements accounted for from P
```

5.1.3 Constructing a Clause

The head of each clause will always be the predicate with variable arguments that are all different. Therefore constructing a clause involves finding a sequence of literals to make up the body. A new literal must either be of the form `X=Y` or `\+ X=Y`, where `X` and `Y` are variables already occurring in the clause, or of the form `p(X1,X2,...Xn)` or `\+ p(X1,X2,...Xn)`, where `p` is any predicate and `X1`, ... `Xn` are either new or existing variables. Here is how a clause is constructed:

```
Let T be the current value of P, together with all the non-examples
Until T contains no non-examples do
    Add a new literal to the end of the clause
    Set T to the elements of T that satisfy the new literal
    Expand T to take into account any new variables in the new literal
```

All that we now need to discuss is how the system chooses a literal to add to the clause.

5.1.4 Selecting a New Literal

FOIL computes for each possible addition to a clause a number called the "gain". Then the addition with the highest gain is chosen. The purpose of a clause is to provide information about which data elements are positive examples of the relation. Therefore an addition to a clause, transforming it from T to T', can be evaluated as promising if it counts high in terms of:

- the number of positive examples accepted by T which are still accepted by T'. I.e. we want to have the final clause subsume as many positive examples as possible and so don't want to add new literals which reduce the number too much.

- the extent to which the ratio of positive examples subsumed to the sum of both positive and negative examples subsumed has gone up in proceeding from T to T'. I.e. we want to head towards a clause that only subsumes positive examples, even if this means not subsuming very many of them.

The actual formula used by FOIL is motivated by considerations of information theory, which we will consider more thoroughly in Chapters 9 and 10 in connection with ID3.

5.1.5 Performance and Problems

There are a number of other features of FOIL that slightly complicate the algorithm presented here, but the main the ideas behind the system are as described. FOIL seems to be able to learn an impressive number of different programs, including (not quite optimal) versions of `append` and `reverse`. However, the system needs a large number of examples for the information-theoretic heuristic to work well (Quinlan uses 10261 data points for learning `append`, whereas Shapiro only uses 34). This is not surprising, as it is essentially a statistically-based heuristic.

The information-theoretic heuristic discriminates between different literals on the basis of their ability to discriminate between positive and negative examples. But not all literals in all definitions have this function - for instance, the `partition` goal in `quicksort` does not, and so FOIL cannot learn `quicksort`. In fact, later work by Quinlan has investigated choosing literals that do not produce any gain but which are *determinate*. Another problem is that once FOIL has picked the wrong literal, it cannot go back and change it. It would be better if FOIL had a search strategy that was less committed, for instance a beam search rather than the hill-climbing approach described. On the other hand, the directedness of the search is one of the good features of FOIL (when it works).

5.2 Top-Down and Bottom-Up Methods

Both MIS and FOIL are “top-down”, in that they generate possible programs and then compare them with the data to be explained. In particular, in generating individual clauses both start with very general clauses and then enumerate possible specialisations of these in a general-specific search. The opposite, “bottom-up”, approach to inductive logic programming involves working from the data to generate the programs directly.

The bottom-up approach to ILP involves taking a set of examples and hypothesising a set of axioms that would produce these theorems by resolution. Just as unification (which, given two terms, produces a term that is more instantiated than each) is the basis for normal resolution, one of the bases for “inverse res-

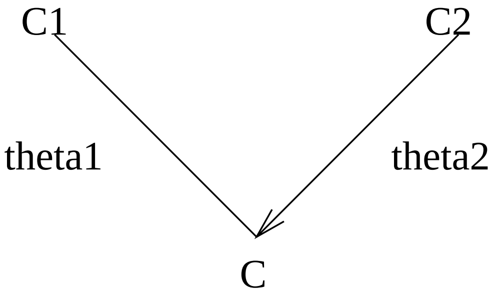


Figure 5.1: V Operator

olution" is *generalisation*, which, given two terms, produces a term which is less instantiated than each.

5.3 Inverting Resolution - CIGOL

CIGOL is a system implemented by Muggleton which uses "inverse resolution" to generate programs from examples. Here we will just consider two of the main operators used by his system and, informally, how they work. One of the pleasant properties of inverse resolution is that it is able to (via the W operator) invent new predicates. Usually the power of a learning system is limited by the initial "vocabulary" of predicates, etc., that is provided to it. Conceptual clustering is one way in which a learning system can invent new vocabulary; inverse resolution seems to be another way of overcoming this barrier.

5.3.1 The V Operator

The standard resolution rule produces from two clauses C1 and C2 the resulting clause C (Fig 5.1). We assume that the literal resolved on appears positively in C1 and negatively in C2. The operation of *absorption* constructs the clause C2, given C1 and C. In fact, in CIGOL this is only implemented for the case where C1 is a unit clause (fact). In this case, the situation is something like that shown in Fig 5.2. So here the clause `D :- A, E` is derived as the result of inversely resolving from `D :- E`, given `A`. Absorption involves carrying out the following procedure:

- Find some instantiation of C1, with substitution theta1.
- Construct a new clause with C, together with this new instance of C1 as an extra goal.
- Generalise the result, to get C2.

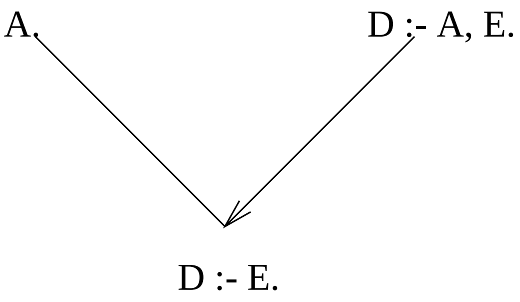
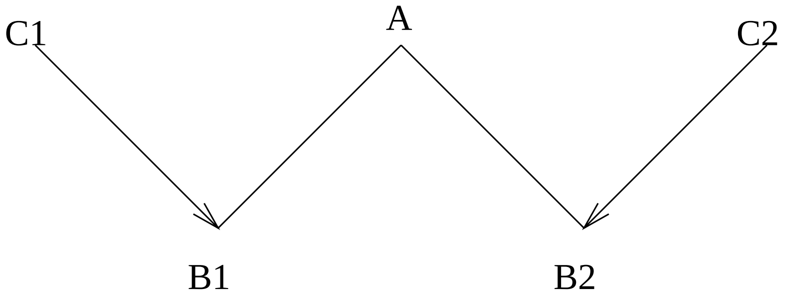


Figure 5.2: V Operator instance

Figure 5.3: The W Operator



For instance, given:

C1 = `(B < s(B))`
C = `(A < s(s(A))`

we can produce the following:

- The instantiation `s(A) < s(s(A))` of C1.
- The new clause `A < s(s(A)) :- s(A) < s(s(A))`
- The generalised clause `A < D :- s(A) < D`

That is, the system has inferred a general principle which accounts for how `A < s(s(A))` follows from `B < s(B)`.

5.3.2 The W Operator

The W operator is involved when a number of clauses B1, B2, ... result from resolving away the same literal L in a clause A when the clause is resolved with C1, C2, ... Figure 5.3 shows this for the case of two results. CIGOL's *intra-construction* produces values for A, C1, C2, given B1 and B2 as inputs. It

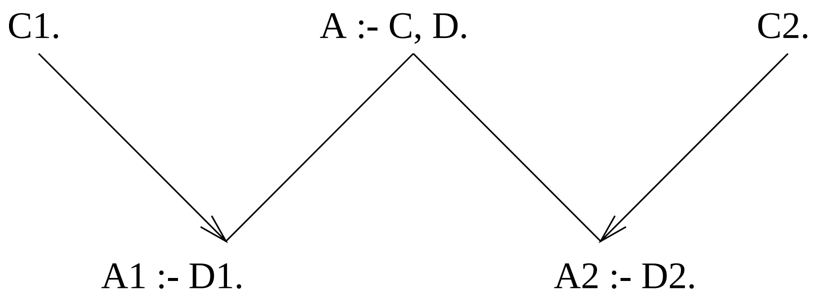


Figure 5.4: W Operator instance

assumes that C1 and C2 are both unit clauses. Thus the situation is something like that shown in Figure 5.4. The clause A is basically a generalised version of B1 and B2 with an extra goal. The two facts C1 and C2 are for the same predicate that this goal has. The approach used in CIGOL is to assume that this is a new predicate. Thus we can carry out intra-construction by the following steps:

- Find a clause B which generalises both B1 and B2. Remember the substitutions theta1 and theta2 that produce B1 and B2 from this generalisation.
- Construct the literal L by taking a new predicate p, together with all the "relevent" variables in the domain of the substitutions theta1, theta2.
- We can then have A be the clause B, with L as an extra goal.
- To ensure that the appropriate substitutions are applied with A is resolved with C1 (and C2), we let:

  C1 = L.theta1
  C2 = L.theta2

Thus for example, if we have:

B1 = `min(D,[s(D)|E]) :- min(D,E).`
B2 = `min(F,[s(s(F))|G] :- min(F,G).`

then we can get the following:

- the generalised clause (B) `min(H,[I|J]) :- min(H,J).`, together with the two substitutions:

  theta1 = {H/D, I/s(D), J/E}
  theta2 = {H/F, I/s(s(F)), J/G}

- the new literal (L) `p(H,I)`.

- the new clause (A) `min(H,[I|J]) :- min(H,J), p(H,I).`
- the new p facts:

  C1 = `p(D,s(D)).`
  C2 = `p(F,s(s(F))).`

What the system has "discovered" here is the "less than" predicate.

5.3.3 Search

Although these examples may look (fairly) simple, a practical system has to decide when to use each operation and also how exactly to do each one. For instance, there will be many clauses that generalise any given two that are provided. Thus although inverse resolution is very elegant there are important search control problems to be solved in order to make it practical.

5.4 References

- Quinlan, J. R., "Learning Logical Definitions from Relations", *Machine Learning* Vol 5, pp239-266, 1990.
- Muggleton, S. and Buntine, W., "Machine Invention of First-Order Predicates by Inverting Resolution", *Fifth International Conference on Machine Learning.* pp339-352, Morgan Kaufman, 1988.

Chapter 6

Classification Learning

The general task of learning a classification is a generalisation of concept learning, in that the result should be able to place an unseen observation into one of a set of several categories, rather than just to pronounce whether it an instance of a concept or not.

6.1 Algorithms for Classification

The following represents a general way of using a concept-learning algorithm for a classification task:

- For each category in turn:
    1. Take the instances of the category as positive examples.
    2. Take the instances of the other categories as negative examples.
    3. Take the generalisations made in the learning of the previous categories also as negative examples.
    4. Apply the concept learning algorithm to this data to get a learned description for this category.

Notice that the third step is necessary to make sure that the learned descriptions for the categories are disjoint (later categories are not allowed to impinge on the areas that the earlier categories have generalised into). This obviously means that the categories dealt with earlier will tend to have more general descriptions than those dealt with later.

If the concept learning algorithm used in this framework is AQ, the result is AQ11, an algorithm used by Michalski and Chilausky in a famous system to disgnose soya bean diseases. The third step of the algorithm is only normally possible if generalisations can be treated in the same way as normal data elements, but this is the case with the operations of Candidate Elimination. In AQ11, the third step involves taking the disjuncts g_i of the learned descriptions and using them as negative examples of the other categories.

In the approach just described, special steps are necessary to make sure that the descriptions of the different categories are disjoint. One way to do this, which is the basis of almost all symbolic approaches to classification, is, rather than representing the different classes separately, to combine their definitions into a *decision tree*. The following program illustrates the use of a decision tree to organise the "learned" information used to identify an animal.

6.2 Demonstration: The 'Animals' Program

This program attempts to guess an animal that you have thought of by asking yes/no questions. If its knowledge is not adequate to find the correct answer, it expands it. The system maintains its knowledge in the form of a discrimination net (decision tree).

The "animals" program is based loosely on the structure of Feigenbaum's EPAM system. EPAM is an attempt to model the behaviour that people exhibit when learning associations between nonsense syllables (e.g learning that the response to the stimulus FAK should be XUM). It builds a discrimination network in essentially the same way as our program, but:

- EPAM creates its own discriminating tests, rather than asking the user.
- Although in EPAM the network is mainly for discriminating between the different possible stimuli, it is also used to associate responses with stimuli. Complete responses are stored in the network as is just enough information about stimuli to discriminate between then. The information associated with the stimulus is just enough information to retrieve the response from the network at the moment of association.

EPAM displays a number of interesting kinds of behaviour that are also found in humans, including stimulus and response generalisation, oscillation and retroactive inhibition and forgetting.

Do we want to call this learning? We will come back to decision trees after looking at numerical approaches to classification. We will also look at the use of discrimination nets in unsupervised learning when we consider conceptual clustering.

6.3 Numerical Approaches to Classification

One way of formulating the solution for a classification problem is to have for each category c_i a function g_i that measures the likelihood that a given $\mathbf{x}$ is in that category. The collection of these functions is called a *classifier.* A classifier assigns an observation $\mathbf{x}$ to category c_i if

$$g_i(\mathbf{x}) > g_j(\mathbf{x}) \text{ for all } j \neq i$$

This approach is mainly relevant where we have numerical data; when the data is partly symbolic or we wish to integrate the result of learning with other systems (e.g. expert systems) then other representations (e.g. decision trees and rule sets) for the solution will be appropriate.

In the next few lectures, we will look at some of the different kinds of discriminant functions/ classifiers that have been used and how they may be calculated from a set of "training" observations. We concentrate on three main types:

1. Functions based on simple "distance" measures (nearest-neighbour learning and case-based learning).
2. Functions based on an assumed probability distribution of observations (Bayesian classifiers).

In Chapters 12 and 13 we will look at reinforcement approaches to classification which attempt to separate the observation space into simple geometrical regions (linear classifiers, leading to connectionist approaches).

6.4 Reading

Thornton Chapter 4 is a description of AQ11, but it is rather misleading in some ways. The Encyclopaedia of AI (Volume 3, pp423-428) has some brief material on AQ11.

- Feigenbaum, E. A., "The Simulation of Verbal Learning Behaviour", in Feigenbaum, E. and Feldman, J., Eds., *Computers and Thought*, McGraw-Hill, 1963.

Chapter 7

Distance-based Models

In this lecture, we look at classifiers/ discriminant functions based on the idea of computing a "distance" between an observation and the set of examples in a given class.

7.1 Distance Measures

In order to talk about how similar different observations are, we need a measure of distance between observations. Two standard notions of distance between two m-dimensional points are the *Euclidean metric* and the *city block (Manhatten) metric*. If the two observations are the first and second samples, then the Euclidean distance is:

$$\sqrt{\sum_{j=1}^{m}(x_{1j} - x_{2j})^2}$$

This corresponds to the "as the crow flies" distance. The Manhatten distance is:

$$\sum_{j=1}^{m} |x_{1j} - x_{2j}|$$

This is the distance we would have to traverse in walking from one point to the other if we were constrained to walk parallel to the axes of the space.

7.2 Nearest neighbour classification

One of the simplest types of classifiers works as follows:

$g_i(\mathbf{x})$ = the distance from $\mathbf{x}$ to its nearest neighbour in class c_i

That is, the classifier will choose for $\mathbf{x}$ the class belonged to by its nearest neighbour in set of training observations. Either of the distance measures introduced above could be used for this.

(Clocksin and Moore 1989) use nearest neighbour classification as part of a system for a robot to learn hand-eye coordination.

7.3 Case/Instance-Based Learning (CBL)

The idea of nearest-neighbour classification can be generalised, to give a class of learning algorithms that are *instance based*, in the sense that no explicit representation is built of the concept/class learned apart from the instances that have been recorded in the "case-base". The general framework for such systems requires something like the following set of functions:

Pre-processor. This formats the original data into a set of cases, possibly normalising or simplifying some of the feature values.

Similarity. This determines which of the stored instances/cases are most similar to a new case that needs to be classified.

Prediction. This predicts the class of the new case, on the basis of the retrieved closest cases. It may just pick the class of the closest one (nearest neighbour), or it may, for instance, take the most frequent class appearing among the k nearest instances.

Memory updating. This updates the stored case-base, possibly adding the new case, deleting cases that are assumed incorrect, updating confidence weights attached to cases, etc. For instance, a sensible procedure may be only to store new cases that are initially classified *incorrectly* by the system, as other cases are to some extent redundant.

7.3.1 Distance Measures

The most difficult part of a CBL system to get right may well be the measure of *distance* to use (equivalently, the notion of *similarity*, which is inversely related to distance). Although the Euclidean and Manhatten metrics can be used for numeric-valued variables, if some variables have symbolic values then the formula needs to be generalised. A common distance formula for the distances between cases numbered i and j would be something like:

$$\Delta(i,j) = \sum_{k=1}^{m} w_k \delta_k(x_{ik}, x_{jk})$$

where m is the number of variables and x_{ik} is the value of variable x_k for the ith case. w_k is a "weight" expressing the relative importance of variable x_k, and δ_k is a specific distance measure for values of x_k. Possible functions for δ_k would be as follows:

Numeric values. The absolute value, or the square, of the numeric difference between the values. But if the range of possible values splits into certain well-defined intervals, it may be better first to determine the relevant intervals and then to apply a distance measure for these as nominal values.

Nominal values. The distance could be 1 if the values are different or 0 if the values are the same. A more sophisticated approach would be something like the following measure (used in the Cost and Salzberg paper referenced below):

$$\delta_k(v_1, v_2) = \sum_{c=1}^{C} \left| \frac{f_k(v_1, c)}{f_k(v_1)} - \frac{f_k(v_2, c)}{f_k(v_2)} \right|$$

Here the sum is over the possible categories, $f_k(v)$ is the frequancy with which variable k has value v in the case base and $f_k(v, c)$ is the frequency with which a case having value v for variable k is assigned the class c. This measure counts values as similar if they occur with similar relative frequencies within each class.

Structured values. If the possible values belong to an abstraction hierarchy, then two values can be compared by computing the most specific concept in the hierarchy which is at least as general as each one. A measure of the distance is then the inverse of a measure of the specificity of this concept (i.e. the more specific the concept that includes both of the values, the more similar the values are.

7.3.2 Refinements

Here are some refinements which have been tried with some success:

- Maintaining weights on the instances in the case-base to indicate how "reliable" they have been in classifying other cases. Using these to selectively remove seemingly "noisy" cases, or to affect the distance measure (less reliable cases seem further away).

- Updating the weights w_k associated with the different variables to reflect experience. When the classification of a new case is known, the updating can be done on the basis of the nearest neighbour(s) and which of their feature values are similar to those of the new case. If the neighbour makes a correct prediction of the class, then the weights for features whose value are similar can be increased and other feature weights decreased. If the neigbour makes an incorrect prediction then the weights for dissimilar features can be increased and the other weights decreased.

7.3.3 Evaluation

The PEBLS system of Cost and Salzberg, which deals with symbolic features and incorporates some of the above refinements, has been compared with connectionist learning schemes and ID3 (a symbolic learning method that builds decision trees) and seems to be very competitive.

7.4 Case Based Reasoning (CBR)

Case-based learning can be seen as a special form of the more general notion of case-based reasoning, which uses a case-base and similarity measures to perform other reasoning tasks apart from classification learning. Case based reasoning (CBR) can be regarded as an alternative to rule-based reasoning that has the following advantages:

- It is more likely to work in badly-understood domains.
- It gets better with practice, but is able to start working very quickly.
- It mirrors some aspects of the way humans attack complex problems.
- It allows new knowledge (i.e. cases) to be integrated with little difficulty.
- The knowledge acquisition process is much easier (though this depends on the complexity of the similarity measure that has to be developed).

The general procedure for CBR is more complex than what we have seen for CBL (though it could be used for more complex learning problems). It goes something like the following:

Assign indices. Indentify and format the features of the current problem, by assigning indices (essentially keywords) to the key features.

Retrieve. Retrieve past cases from the case base with similar indices.

Adapt. Adapt the solution stored with the retrieved case to the new situation.

Test. See whether the proposed solution works, and update various knowledge sources accordingly.

In CBR, indices need not correspond directly to variables and values in the way that we have considered for learning. In computing distances, it may be non trivial to determine which indices of one case correspond to which in another. Distance may be judged in terms of complex criteria such as the goodness of a chain of inferences between two indices. Parts of cases may be entered as free text, and it will be necessary to compute the distance between these (perhaps by using word or n-gram frequencies). Attention may need to be paid to the way that memory is organised in order to facilitate rapid retrieval of close cases when the format of cases is flexible.

In general, though perhaps not on average, the similarity calculation in a CBR system may be as complex as a set of rules in a rule-based system. Nevertheless, CBR has proved an attractive technology and has produced impressive applications. Several CBR shells are now available commercially, using simple distance metrics that can be tailored by the user. One interesting class of applications is providing "retrieval only" services for advisory services (e.g. help desks). Here the facility to introduce (partly) free text descriptions of problems and to retrieve descriptions of previous and related cases, perhaps via a focussed question-and-answer dialogue, has been very valuable Compaq received 20% less calls to their customer support centre when they supplied a CBR system (QUICKSOURCE) to their customers to help them with common printer problems (a saving of over $10M per year).

7.5 Background Reading

Standard mathematical distance measures are discussed in Manly Chapter 4 and also in Beale and Jackson, section 2.6. The two papers by Aha give a good summary of CBL approaches. The Cost and Salzberg paper describes the PEBLS system, which incorporates techniques for dealing with symbolic values. Kolodner is one of the standard books on CBR.

- Aha, D., "Case-Based Learning Algorithms", in Procs of the DARPA Case-Based Reasoning Workshop, May 1991, Morgan Kaufmann publishers.
- Aha, D., Kibler, D. and Albert, M., "Instance-Based Learning Algorithms", *Machine Learning* Vol 6, No 1, pp37-66, 1991.
- Beale, R. and Jackson, T., *Neural Computing: An Introduction*, IOP Publishing, 1991, Chapter 2.
- Clocksin, W. F. and Moore, A. W., "Experiments in Adaptive State-Space Robotics", Procs of the Seventh Conference of the SSAISB, Pitman, 1989.
- Cost, S. and Salzberg, S., "A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features", *Machine Learning* Vol 10, pp57-78, 1993.
- Kolodner, J., *Case-Based Reasoning*, Morgan Kaufmann, 1993.
- Manly, B. F. J., *Multivariate Statistical Methods*, Chapman and Hall, 1986.

Chapter 8

Bayesian Classification

So far, we have looked at the appropriateness of mathematical methods based on the idea of seeing possible observations as points in an n-dimensional space. In reality, however, a concept is not just a subspace (set of points), but has associated with it a particular probability distribution. That is, not all observations are equally likely. In this lecture, we consider techniques that are statistically based, i.e. they take account of the fact that observations come from underlying probability distribution.

8.1 Useful Statistical Matrices and Vectors

Recall that x_{ij} is the ith measurement of variable x_j. There are n observations of this variable given. Then the *sample mean* of variable x_j, written $\bar{x_j}$, is defined as follows:

$$\bar{x_j} = \frac{\sum_{i=1}^{n} x_{ij}}{n}$$

Each $\bar{x_j}$ gives the average measurement in a different dimension. If we put these together into a vector, we get the following as the overall sample mean:

$$\bar{\mathbf{x}} = \begin{pmatrix} \bar{x_1} \\ \bar{x_2} \\ \vdots \\ \bar{x_n} \end{pmatrix}$$

$\bar{x}$ can be regarded as a point in the same way as all the observations. Geometrically, it represents the "centre of gravity" of the sample.

Whilst the mean defines the "centre of gravity", covariances measure the variation shown within the sample. If x_j and x_k are two variables, their *coveriance* within the sample is:

$$covariance(x_j, x_k) = \frac{\sum_{i=1}^{n}(x_{ij} - \bar{x_j})(x_{ik} - \bar{x_k})}{(n-1)}$$

This is a measure of the extent to which the two variables are linearly related (correlated). This sum will be large and positive if samples of x_j which are greater than the $\bar{x_j}$ correspond to samples of x_k which are greater than $\bar{x_k}$ and similarly with samples less than the mean. If samples of x_j greater than $\bar{x_j}$ correspond to samples of x_k less than $\bar{x_k}$ then the value will be large and negative. If there are no such correlations, then the positive and negative elements in the sum will cancel out, yielding a covariance of 0. It is useful to collect the covariances of a sample into an $m \times m$ matrix $\mathbf{C}$, as follows:

$$\mathbf{C}_{jk} = covariance(x_j, x_k)$$

As a special case of covariance, the *sample variance* of the variable x_j is the covariance of x_j with itself:

$$var(x_j) = covariance(x_j, x_j)$$

This is a measure of the extent to which the sample values of x_j differ from the sample mean $\bar{x_j}$. The square root of the variance is the *standard deviation.* Note that if the means $\bar{x_j}$ of the variables are standardised to zero (by subtracting the mean from each value), then

$$covariance(x_j, x_k) = \frac{\sum_{i=1}^{n} x_{ij} x_{ik}}{(n-1)}$$

and so in fact

$$\mathbf{C} = \frac{1}{(n-1)} \sum_{observations\ \mathbf{x}} \mathbf{x}\mathbf{x}^t \tag{8.1}$$

8.2 Statistical approaches to generalisation

Statistical approaches involve fitting models to observations in the same way as other mathematical approaches. In statistics, this activity is often called *data compression.* A statistical model can be used to generalise from a set of observations in the following way:

- A model is selected (e.g. it is decided that the number of milk bottles appearing on my door step every morning satisfies a Normal distribution).

- The closest fit to the data is found (e.g. the best match to my observations over the last year is a distribution with mean 3 and standard deviation 0.7).

- The goodness of fit with the data is measured. With a statistical model, one may often be able to evaluate the probability of the original data occurring, given the chosen model).

- If the fit is good enough, the model is applied to new situations (e.g. to determine the probability that tomorrow there will be four bottles on my doorstep).

A major advantage of almost all the methods is that they come with a way of measuring error and significance. This does not apply to such an extent with symbolic models of learning. Although error functions can be somewhat arbitrary (and can be devised for symbolic, as well as numerical, learners), many of the statistical methods will actually give a *probability* indicating how likely it is that the model applies, how likely it is that a given observation is an instance of the concept, etc. This is a real bonus - for instance, it can be used to determine what the risks are if you act on one of the predictions of the learner.

8.3 Example: Multivariate Normal Distribution

The Normal Distribution has a special significance for studies of a single random variable, not least because of the Central Limit Theorem, which states that means taken from samples from any random variable (with finite varience) tend towards having a Normal distribution as the size of the sample grows.

The generalisation of the Normal Distribution to the situation where there are m variables is the *multivariate normal distribution.* In the multivariate normal distribution, the probability of some observation $\mathbf{x}$ occurring is as follows:

$$P(\mathbf{x}) = \frac{1}{(2\pi)^{m/2}|\mathbf{C}|^{1/2}} e^{-1/2(\mathbf{x}-\mu)^t \mathbf{C}^{-1}(\mathbf{x}-\mu)} \tag{8.2}$$

where m is the number of dimensions, $\mathbf{C}$ the covariance matrix and μ the mean vector. The details of this formula do not matter - the important point is that if a population are in a multivariate normal distribution, the significant aspects of that population can be summed up by μ and $\mathbf{C}$.

If a sample does come from a multivariate normal distribution, the best estimates for the mean and covariance matrices for the population are those calculated from the sample itself (using the formulae above). Given these and the formula for $P(\mathbf{x})$, it is possible to calculate the probability of $\mathbf{x}$ occurring within any given region, and hence the expected number of observations in this region for a sample with the size of the training sample. Then the formula:

$$\sum_{region_i} \frac{(observed\ occurrences\ in\ region_i -\ expected\ occurrences\ in\ region_i)^2}{expected\ occurrences\ in\ region_i}$$

(where the regions $region_i$ are mutually exclusive and exhaustive) gives a measure of the discrepancy between the observed sample and what would be expected if it was multivariate normal. The value of this sum can be used to determine the probability that the sample is indeed multivariate normal (this is called the *chi-square test*).

8.4 Using Statistical "distance" for classification

A variation on nearest neighbour classification would be to measure the distances from the new observation to the *means* of the different classes, selecting the class whose mean was closest to the observation. As with nearest neighbour classification, however, this has the problem that it does not take adequate account of:

- The fact that some populations have more 'scatter' than others
- The fact that other factors may affect the probability of being within a given class (e.g. it may be known that one class only contains very rare cases).

Bayesian classification is an alternative that gets over these problems.

8.5 Bayesian classification

Linear classification and nearest neighbour classification can both be criticised for ignoring the probability distributions of the training observations for the different classes. Bayesian classification is one way of improving this.

Given an observation $\mathbf{x}$, classification requires a measure of likelihood that $\mathbf{x}$ belongs to each class c_i. A natural way to get this measure is to estimate $P(c_i|\mathbf{x})$, the probability that c_i is observed, given that $\mathbf{x}$ is observed. Bayes rule then gives:

$$g_i(\mathbf{x}) = P(c_i|\mathbf{x}) = \frac{P(\mathbf{x}|c_i)P(c_i)}{P(\mathbf{x})}$$

If we have already established a plausible probability distribution for the population of examples of c_i, then this formula may be straightforward to evaluate.

Although we might want to use this formula directly if we wanted to get an absolute picture of how likely $\mathbf{x}$ is to be in c_i, if we only want to use the formula to compare the values for different i, then we can ignore the $P(\mathbf{x})$ term and apply monotonically increasing functions to it without affecting any of the decisions made. If we apply the logarithm function, we get:

$$g_i(\mathbf{x}) = log(P(\mathbf{x}|c_i)) + log(P(c_i))$$

Now, if we have already fitted a multivariate normal distribution (with mean μ_i and covariance matrix $\mathbf{C}_i$) to the examples in c_i then this formula is easy to evaluate. Substituting in equation 8.2, and removing constants from the sum, we have:

$$g_i(\mathbf{x}) = -(1/2)log(|\mathbf{C}_i|) - (1/2)(\mathbf{x} - \mu_\mathbf{i})^t{\mathbf{C}_i}^{-1}(\mathbf{x} - \mu_\mathbf{i}) + log(P(c_i))$$

If we assume that the covariance matrix $\mathbf{C} = \mathbf{C}_i$ is the same for all the categories, we can ignore the first term (which is the same for all categories) and use the classification function:

$$g_i(\mathbf{x}) = -(1/2)(\mathbf{x} - \mu_\mathbf{i})^t\mathbf{C}^{-1}(\mathbf{x} - \mu_\mathbf{i}) + log(P(c_i))$$

The quantity

$$(\mathbf{x} - \mu_\mathbf{i})^t\mathbf{C}^{-1}(\mathbf{x} - \mu_\mathbf{i})$$

is called the *Mahalanobis distance* from $\mathbf{x}$ to the population of the c_i examples. If all the c_i are equally probable and the covarience matrices are the same, then its negation on its own provides a good classification function for c_i.

8.6 Advantages and Weaknesses of Mathematical and Statistical Techniques

Most of the mathematical methods incorporate some robustness to error in the original data. Assuming that there is enough data, the significance of a few very odd observations will not be great, given the kinds of averaging processes that take place (this does not apply to nearest-neighbour classification). Symbolic learning systems tend not to have this kind of robustness (of course, it is much easier to average numbers than symbols). On the other hand, most of the mathematical methods are really meant to be applied in a "batch", rather than in an "incremental" mode. This means that, if we want a learner that is accepting a stream of inputs and can apply what it has as it goes along, these systems will have to recompute a lot every time a new input comes in (this does not apply to some uses of gradient descent, as we saw).

One real deficiency of all the mathematical methods is that the inputs have to be numerical. If we wish to deal with symbol-valued attributes, such as colour, then we either have to code the values as numbers (for colours, one could probably use the appropriate light frequencies) or to have a separate 0-1 variable for each possible value (red, blue, etc). In addition, the assumption is made that there is a finite number of dimensions along which observations vary. This may often be the case, but is not always so. In particular, inputs may be recursively structured, in which case there are infinitely many possible dimensions (consider inputs that are binary trees labelled with 1s and 0s, for instance).

8.7 Background Reading

Matrices, means and covariances are discussed in Manly Chapter 2. Regression is discussed in Ehrenberg, Chapter 12 and Chapter 14. Bayesian classifiers are described in Duda and Hart. Manly (chapter 4) discusses the Mahalanobis distance and alternatives to it.

- Duda, R. and Hart, P., *Pattern Classification and Scene Analysis*, Wiley, 1973.
- Ehrenberg, A. S. C., *A Primer in Data Reduction*, John Wiley, 1986.
- Manly, B. F. J., *Multivariate Statistical Methods*, Chapman and Hall, 1986.

Chapter 9

Information Theory

In the discussion of Quinlan's FOIL, we mentioned the role of Information Theory in the heuristic choice of a literal to add to a clause. Information Theory also plays a significant role in the operation of the ID3 family of symbolic classification systems and so it is time to spend some time on it now.

9.1 Basic Introduction to Information Theory

The basic scenario is that of a sender, who may send one of a number of possible messages to a receiver. The information content of any particular message is a measure of the extent to which the receiver is "surprised" by getting it.

Information content obviously relates inverse to probability – the more probable something is, the less surprised one is when it occurs. But also information content may be subjective.

The standard measure for the information content of a message m, $i(m)$ is:

$$i(m) = -\log_2(P(m))$$

This gives a number of *bits*.

We have argued the inverse relation to probability, but why the $\log_2$? Here are some arguments:

Adding information. If m_1 and m_2 are independent messages, then we would like:

$$i(m_1 \wedge m_2) = i(m_1) + i(m_2)$$

The "binary chop" algorithm. The amount of information conveyed = the amount of uncertainty there was before = the amount of work needed to resolve the uncertainty. If there are n equally likely books in alphabetical order, the amount of work needed to locate any one by the algorithm is less that $\log_2(n)$.

Entropy. The logarithm is justified by arguments about entropy – see the next section.

9.2 Entropy

Entropy is a measure of the uncertainty in a "situation" where there is a whole set of possible (exclusive and exhaustive) messages m_i with $\sum_i P(m_i) = 1$. The entropy H is some function of all the probabilities, $H(P(m_1), P(m_2), \ldots P(m_n))$. How should this behave?

- It should be a continuous function of all the $P(m_i)$ (i.e. a small change in the probabilities should lead to a small change in the entropy).
- If the probabilities are all equal, H should increase as n, the number of possible messages, increases.
- It should behave appropriately if a choice is broken down into successive choices. For instance, if there are messages with probabilities $\frac{1}{2}$, $\frac{1}{3}$ and $\frac{1}{6}$, then the entropy should be the same as if there are two messages with probabilities $\frac{1}{2}$ and the first of these is always followed by one of two messages with probabilities $\frac{2}{3}$ and $\frac{1}{3}$. That is,

$$H(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}) = H(\frac{1}{2}, \frac{1}{2}) + \frac{1}{2} H(\frac{1}{3}, \frac{2}{3})$$

It is a *theorem* that the only possible such H is of the form:

$$H = -k \sum_i P(m_i) \log_2(P(m_i))$$

Choosing a value for k amounts to selecting a unit of measure – we will choose $k = 1$.

Consequences of this are:

- If all messages are equally likely,

$$\begin{aligned} H(\ldots) &= -\sum P(m_i) \log_2(P(m_i)) && (9.1) \\ &= -(\sum P(m_i)) \log_2(P(m_i)) && (9.2) \\ &= i(m_i) && (9.3) \end{aligned}$$

  which is what one might hope.

- $H = 0$ only when one $P(m_i)$ is 1 and the others are 0. Otherwise $H > 0$.
- For a given n, H is a maximum when each $P(m_i)$ is $\frac{1}{n}$.

9.3 Classification and Information

If we have a new object to be classified, there is initial uncertainty. We can ask: “Which of the following partial descriptions of the categories reduces this uncertainty most?”, i.e. which parts of the object’s description are most useful for the classification. This is the idea used in ID3.

9.4 References

See Thornton, Chapter 5. The classic reference on information theory is the book by Shannon and Weaver.

- Shannon, C. and Weaver, W., *The Mathematical Theory of Information*, University of Illinois Press, 1949.

Chapter 10

ID3

The Candidate Elimination Algorithm takes an exhaustive and incremental approach to the problem of concept learning. Members of the ID3 family of classification learning algorithms have the following features, which are in contrast to the above.

- They are heuristic. Firstly, there is no guarantee that the solution found is the "simplest". Secondly, there is no guarantee that it is correct – it may explain the data provided, but it may not extend further.
- They are non-incremental. That is, all the data – and plenty of it too, if the numerical heuristics are to be reliable – must be available in advance.
- They make no use of world knowledge. There is no way to use extra knowledge to influence the learning process.

The above characteristics are basically the same as for the FOIL system, which was developed by the same person, though after ID3.

10.1 Decision Trees

ID3 (Quinlan 1986, though he reported it in papers as far back as 1979) is a symbolic approach to classification learning. Quinlan saw machine learning as a way of solving the "knowledge acquisition bottleneck" for expert systems. Thus he was interested in learning representations that could be translated straightforwardly into expert system rules. ID3 learns to classify data by building a decision tree. Figure 10.1 shows an example decision tree that would enable one to predict a possible future weather pattern from looking at the value of three variables describing the current situation - *temperature*, *season* and *wind*.

A decision tree can be translated into a set of rules in disjunctive normal form by traversing the different possible paths from the root to a leaf. In this case, the rules would include:

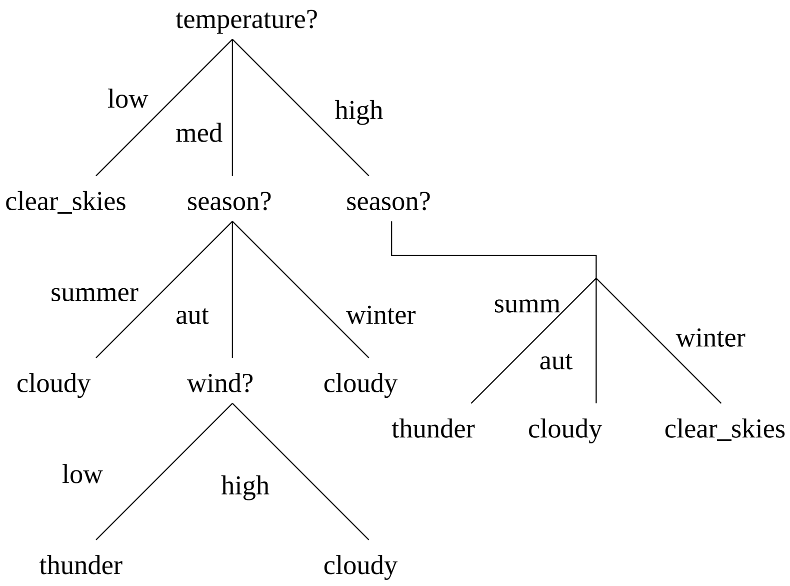

temperature?
low
med
high
clear_skies
season?
season?
summer
aut
winter
summ
aut
winter
cloudy
wind?
cloudy
thunder
cloudy
clear_skies
low
high
thunder
cloudy

Figure 10.1: Decision Tree

```
IF (temperature=high AND season=summer) OR
   (temperature=medium AND season=autumn AND wind=low)
THEN thunder

IF (temperature=high AND season=winter) OR
   (temperature=low)
THEN clear_skies
```

ID3 assumes a set of pre-classified data. There is a finite set of variables and each element specifies a value for each variable. The basic ID3 algorithm assumes symbolic, unstructured values for the variables, though improved algorithms allow other kinds of values.

10.2 CLS

ID3 is based on the CLS algorithm described by Hunt, Marin and Stone in 1966. The CLS algorithm defines a procedure *split(T)* which, given a training set T builds a decision tree. It works as follows:

- If all the elements of T have the same classification, return a leaf node with this as its label.
- Otherwise,
  1. Select a variable ("feature") F with possible values v_1, v_2, $\ldots v_N$.

2. Partition T into subsets T_1, T_2, $\ldots T_N$, according to the value of F.
3. For each subset T_i call *split(T_i)* to produce a subtree $Tree_i$.
4. Return a tree labelled at the top with F and with as subtrees the $Tree_i$s, the branches being labelled with the v_is.

Note that the complexity of the tree depends very much on the variables that are selected.

10.3 ID3

ID3 adds to CLS:

- A heuristic for choosing variables, based on information theory.
- "Windowing" – an approach to learning for very large data sets.

10.3.1 The Information Theoretic Heuristic

- At each stage, calculate for each variable X the expected information gained (about the classification) if that variable is chosen.
- Select the variable X with the highest score.

This is a heuristic, hill-climbing search.

Information gained ($gain(X)$) = Expected information needed (entropy) after - information needed (entropy) before.

Information needed before =

$$\sum_{c_i} -P(c_i)\log_2(P(c_i))$$

where c_1, c_2, etc. are the different categories and the probabilities are estimated from the original (unsplit) population of data elements.

Information needed after =

$$\sum_{v_j} P(v_j). \text{ Info needed for subset } T_j$$

where T_j is the subset arising for value v_j for variable X. This is:

$$\sum_{v_j} \frac{\text{No of elements with } v_j}{\text{Total no of elements}} \sum_{c_k} -P(c_k)\log_2(P(c_k))$$

where the probabilities for the subtrees are estimated from the subpopulations of the data assigned to those subtrees.

10.3.2 Windowing

When there is a huge amount of data, learning will be slow. Yet probably the same rules could be learned from a smaller, “representative”, sample of the data. Windowing works in the following way:

1. Choose an initial window from the data available.
2. Derive a decision tree for this set.
3. Test the tree on the remainder of the data.
4. If exceptions are found, modify the window and repeat from step 2.

The window can be modified in a number of ways, for instance by:

- Adding randomly selected exceptions to the window.
- Adding randomly selected exceptions, but keeping the window size constant by dropping “non-key” examples.

Opinions differ on the utility of windowing.

10.4 Some Limitations of ID3

- The scoring system is only a heuristic – it can’t guarantee the “best solution”.
- It tends to give preference to variables with more than 2 possible values. It is fairly easy to see why this is.
- The rule format has some limitations – it can’t express “if age between 10 and 12” or “if age=10 ,... otherwise ...”, for instance.

Some of these limitations are relaxed in more recent systems, in particular C4.5 which we will consider in the next chapter.

10.5 References

Thornton, Chapter 6.

- Quinlan, J. R., “Induction of Decision Trees”, *Machine Learning* Vol 1, pp81-106, 1986.

Chapter 11

Refinements on ID3

In this chapter we will concentrate on some of the refinements that have been made to ID3, focussing on Quinlan's C4.5. We also present some work that has attempted experimentally to compare the rsults of different classification learning systems.

11.1 The Gain Ratio Criterion

As we pointed out, ID3 prefers to choose attributes which have many values. As an extreme case, if each piece of training data had an attribute which gave that item a unique name then this attribute would always be a perfect choice according to the criterion. But of course it would be useless for unseen test data. In this case, just knowing which subset the attribute assigns a data item to already conveys a huge amount of information about the item. We need to find attributes X which have high gain $gain(X)$ (calculated as before) but where there is not a large information gain coming from the splitting itself. The latter is given by the following:

$$split\ info(X) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} \times \log_2(\frac{|T_i|}{|T|})$$

where the T_i are the subsets corresponding to the different values of X. To take both into account, C4.5 uses their ratio:

$$gain\ ratio(X) = \frac{gain(X)}{split\ info(X)}$$

as the heuristic score used to select the "best" variable X.

11.2 Continuous Attributes

Where attributes have continuous values, ID3 will produce over-specific tests that are dependent on the precise values in the training set. C4.5 allows for the generation of binary tests ($value \leq threshold$ vs $value > threshold$) for attributes with continuous values. C4.5 investigates each possible value occurring in the training data as a possible threshold; for each one the gain ratio can be computed and the best possibility can be compared with those arising from other attributes.

11.3 Unknown Values

In real data, frequently there are missing values for some attributes. If all observations with missing values are discarded, there may not be enough remaining to be a representative sample. On the other hand, a system that attempts to deal with missing values must address the following questions:

- How should the gain ratio calculation, used to select an attribute to split on, take unknown values into account?
- Once an attribute has been selected, which subset should a data item with an unknown value be assigned to?
- How is an unseen case to be dealt with by the learned decision tree if it has no value for a tested attribute?

11.3.1 Evaluating tests

If the value of an attribute is only known in a given proportion, F, of cases, then the information gain from choosing the attribute can be expected to be 0 for the rest of the time. The expected information gain is only F times the change in information needed (calculated using the data with known values), because the rest of the time no information is gained. Thus:

$$gain(X) = F \times (informationneededbefore - informationneededafter)$$

where both sets of information needed are calculated using only cases with known values for the attribute.

Similarly the definition of $split\ info(X)$ can be altered by regarding the cases with unknown values as one more group.

11.3.2 Partitioning the training set

C4.5 adopts a probabilistic approach to assigning cases with unknown values to the subsets T_i. In this approach, each subset is not just a set of cases, but is a set of fractions of cases. That is, each case indicates with what probability it belongs to each given subset. Previously any case always belonged to one subset with probability 1 and all the rest with probability 0; now this has been generalised.

If a case has an unknown value for a chosen attribute, for each possible value the probability of a case in the current situation having that value is estimated using the number of cases with that value divided by the total number of cases with a known value. This probability is then used to indicate the degree of membership that the case has to the subset associated with the given value.

In general, now that subsets contain fractional cases, any calculation involving the size of a set has to take the sum of the probabilities associated with the cases that might belong to it.

11.3.3 Classifying an unseen case

If an unseen case is to be classified but has an unknown value for the relevant attribute, C4.5 explores all possibilities. Associated with each possible subtree is the probability of a case having that value, on the basis of the original cases that were used in training this part of the tree and which had known values. For each path down the tree which could apply to the unseen case, the probabilities are multiplied together. The result is a set of possible outcomes with their probabilities. Multiple paths leading to the same outcome have their probabilities added together and at the end one then has for each possible outcome the combined probability of that applying to the unseen case.

11.4 Pruning

Especially if the data is noisy, ID3 can grow an excessively complex tree which *overfits* the training data and performs badly on unseen data. The idea of pruning in C4.5 is to remove parts of the tree whose complexity is not motivated by the extra performance they give. C4.5 prunes its trees in the following ways:

- By discarding a whole subtree and replacing it by a leaf (expressing the class associated most often with the subtree).
- By replacing a subtree by one of its branches (the most frequently used one).

C4.5 uses a heuristic measure to estimate the error rate of a subtree. It does this by assuming that the cases it has been trained on are a random sample from a distribution with a fixed probability of misclassification. If there are N cases covered of which E are misclassified (E will be zero for part of a tree built before pruning), it determines the highest value the misclassification probability could be such that it would produce E misclassifications from N cases with a probability greater than some threshold. A subtree is then replaced by a leaf or a branch if its heuristic misclassification probability is higher. The pruning process works up the tree from the leaves until it reaches a point where further pruning would increase the predicted misclassification probability.

11.5 Converting to Rules

The simplest way to translate a decision tree to rules is to produce a new rule for each path through the tree. Although the resulting rules correctly express what is in the tree, many rules contain unnecessary conditions, which are implied by other conditions or unnecessary for the conclusion of the rule to hold. This arises because the tree may not capture generalisations that can only be seen by putting together distant parts. The result is that the rules are often undigestible for human beings.

C4.5 has heuristics to remove redundant conditions from rules (by considering the expected accuracy with the condition present and absent). For each class it removes rules for that class that do not contribute to the accuracy of the set of rules as a whole. Finally it orders the rules and chooses a default class.

11.6 Windowing

C4.5 provides an option to use windowing, because it can speed up the construction of trees (though rarely) and (with an appropriately chosen initial window) lead to more accurate trees. C4.5 enhances the windowing approach used in ID3 by:

- Choosing an initial window so that "the distribution of classes is as uniform as possible". I'm not sure exactly what this means.

- Always including at least half of the remaining exceptions in the window at each stage (whereas ID3 had a fixed ceiling) in an attempt to speed convergence.

- Stopping before all the exceptions can be classified correctly if the trees seem not to be getting more accurate (cf the discussion of pruning above).

11.7 Grouping Attribute Values

C4.5 provides an option to consider groups of attribute values. Thus instead of having the tree branching for values v_1, v_2, ... v_n it could, for instance, build a tree with three branches for (v_1 or v_2), (v_3) and (all other values). When this option is selected, having just split a tree C4.5 considers whether an improvement would be gained by merging two of the subsets associated with new branches (using the gain ratio or simple gain criterion). If so, the two branches producing the best improvement are merged. Then the same procedure is repeated again, until no further improvement can be reached. Finally the subtrees are recursively split as usual.

As with the other uses of the gain ratio and gain criteria, this is a heuristic approach that cannot be guaranteed to find the best result.

11.8 Comparison with other approaches

Now that we have considered a number of approaches to classification, we can consider how they match up against each other. Indeed there have been a number of interesting experiments attempting to do this.

Initial comparisons carried out in the literature suggest that:

- Rule-oriented learning is much faster than connectionist learning (for this kind of task) and no less accurate.
- Rule-oriented learning can achieve as good results as statistical methods (and, of course, the results are also more perspicuous).

Note, however, that detailed comparison is fraught with problems. In particular, the best algorithm seems to depend crucially on properties of the data. For instance, King et al. found that symbolic learning algorithms were favoured when the data had extreme statistical distributions or when there were many binary or nominal attributes.

There are many problems with comparing different learning algorithms.

- Many algorithms use numerical parameters and it may take an expert to "tune" them optimally.
- Often there are different versions of systems and it is unclear which one to use.
- It is necessary to find large enough datasets to get significant results, and artificially created datasets may not give realistic results.
- It is hard to measure learning speed in a way not distorted by differences in hardware and support software.

There are three main measures of the quality of what has been learned.

1. The percentage of the *training data* that is correctly classified. Not all learning systems build representations that can correctly classify all the training data. But obviously a system that has simply memorised the training data will do perfectly according to this score.
2. The percentage of some *test data* that is correctly classified. Here it is necessary to put some of the available data (usually 30%) aside in advance and only train on the rest of the data. The problem with this is deciding what should be the test data – one could pick a subset that is rather unrepresentative of the set of possibilities as a whole.

3. Using *cross validation*. This attempts to overcome the problems of the last method. The idea is to split the data into n equal sized subsets. The learning system is trained on the data in all n subsets apart from the first and then tested on the remaining subset. Then it is trained on all subsets apart from the second and tested on the second. And so on, n times. The average of the n performances achieved is then taken as a measure of the overall performance of the system.

11.9 Reading

The description of C4.5 follows the presentation in Quinlan's book very closely. Mooney et al, Weiss and Kapouleas and King et al. describe comparative experiments on different types of classification systems.

- Quinlan, J. R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.
- Mooney, R., Shavlik, J., Towell, G. and Grove, A., "An Experimental Comparison of Symbolic and Connectionist Learning Algorithms". In *Readings in Machine Learning*.
- Weiss, S. M. and Kapouleas, I., "An Empirical Comparison of Pattern Recognition, Neural Nets and Machine Learning Classification Methods", Procs of IJCAI-89 (also in *Readings in Machine Learning*).
- King, R. D., Feng, C. and Sutherland, A., "STATLOG: Comparison of Classification Algorithms on Large Real-World Problems", *Applied Artificial Intelligence* Vol 9, No 3, 1995.

Chapter 12

Reinforcement Learning

12.1 Demonstration: Noughts and Crosses

This is a program that learns to play noughts and crosses by playing games, rewarding moves that are made in winning games and penalising moves that are made in losing games. To run it, do the following:

```
% export ml=~dai/courses/ai3-4/machine_learning
% sicstus
% ['$ml/lib/noughts'].
```

To play a game (and have the system update its recorded scores accordingly), call the predicate `game`.

The program is similar to a machine (called MENACE) built by Michie and Chambers using matchboxes and coloured beads. Similar (and more sophisticated) systems have been used by Mitchie and Chambers, and by Clocksin and Moore, for robot control tasks.

This program follows the general pattern of a reinforcement learner, as introduced in Section 1.4.1. That is, the system cycles through getting new training examples, evaluating its performance on them and revising its internal representation in order to do better next time. In a system of this kind, there is a tradeoff between immediate performance and the collection of useful information for the future (*exploitation* vs *exploration*). It is also very important which examples the system is trained on. In this case, if the program always plays against a weak player then it will never get experience in responding to good moves.

12.2 Reinforcement and Mathematical approaches to generalisation

In the next couple of lectures we will consider one major application of reinforcement learning – fitting mathematical models to data. We consider the case where the behaviour of a system is determined by a mathematical model of some kind, which depends on a set of numerical parameters. The task is to learn the values of the parameters that give the "best fit" to the training data. A mathematical model can be used to generalise from a set of observations in the following way:

- It is determined which variables are involved aand which ones need to be inferred from which others (e.g. x and y are the variables and y needs to be inferred from x).
- A model is selected (e.g. $y = ax + b$). This is essentially providing the learner with a "bias").
- The closest fit to the data is found (e.g. $a = 1, b = 4$).
- The model is applied to new situations (e.g. $x = 5$ gives $y = 9$).

Once it has been determined that a particular model fits the data well, applying this model to generate a new point amounts to a kind of *interpolation* from the given points.

A common numerical technique for looking for the closest fit, gradient descent, can be viewed as a kind of reinforcement learning. Initially a guess is made about what values the numerical parameters should have. Then it is seen how the model performs on some training data. According to the kinds of errors made, the parameters are adjusted. And the cycle is repeated, until the parameters have stabilised to a point where performance is good (it is hoped).

12.3 Gradient Descent

Gradient descent is a frequently used method for "learning" the values of the parameters in a function that minimise the "distance" from a set of points that are to be accounted for. It is a kind of hill-climbing search for the function that produces the smallest errors on the data it is supposed to account for. Since it is a numerical technique used in a number of situations, we will spend a little time on it here.

What happens in general is that form of the function to be learned is determined in advance, for instance, it might be determined that it is to be a function of the form $y = ax + b$. The problem then is to find the "best" values for the parameters in this formula (here, a and b). The next step is therefore to define an error function E that enables one to measure how far away a candidate function is from the ideal. For instance, if we required our function to give the values 5, 6 and 7 respectively for x having the values 1, 2 and 3, then the following function would provide a measure of how bad a candidate function was:

$$E = (a + b - 5)^2 + (2a + b - 6)^2 + (3a + b - 7)^2$$

What we have done is add together error terms for the three desired points - each term being the square of the difference between the desired y and the one that the formula would calculate. We now try to find values of a and b that minimise the value of E. In this example, the error is expressed as a sum of squares, and so this is called *least squares fitting*.

The value of E depends on the parameters of the formula (here, a and b). In this case (because there are two parameters) we can think of the error as a surface hovering over a plane corresponding to the different possible values of (a, b), the height of the surface being the value of E. Although this is a relatively simple case, it does give sensible intuitions about the general case. If we pick some initial values of a and b, how can we alter them to find better values that give a smaller error? Geometrically, what we do in gradient descent is to find the direction in which the gradient of the surface downwards is greatest and move the parameters in that direction. The direction picked will have a component in the a direction and a component in the b direction. In general, if p is one of the parameters, then the upwards gradient in the p direction is $\frac{\partial E}{\partial p}$. In order to approximate finding the steepest direction downwards, we adjust each parameter p by an amount proportional to this gradient:

$$p(t+1) = p(t) - \rho \frac{\partial E}{\partial p}$$

where ρ is some constant to be chosen in advance (often known as the "gain"). In this example,

$$\frac{\partial E}{\partial a} = 2(a+b-5) + 4(2a+b-6) + 6(3a+b-7)$$

i.e.

$$\frac{\partial E}{\partial a} = 28a + 12b - 76$$

and the b case is fairly similar. Thus we have:

$$a(t+1) = a(t) - \rho(28a(t) + 12b(t) - 76) b(t+1) = b(t) - \rho(12a(t) + 6b(t) - 36)$$

The gradient descent procedure is then to pick initial values for the parameters and then use these equations repeatedly to compute new values. The iteration will stop when, for instance, E reaches an acceptable level or the incremental changes to p get below a certain size. In this case, if we start (a, b) at $(0, 0)$ and choose $\rho = 0.05$ then a situation with near zero error ($a = 1, b = 4$ approx) is reached in 100 iterations.

Gradient descent can get into problems, for instance if ρ or the initial values for the parameters are chosen badly. The procedure can diverge or get stuck in a local minimum. Sometimes it is possible to prove theorems about the convergence of the procedure.

12.4 Batch vs Incremental Learning

In a learning situation, it is often the case that we are looking for a function that somehow summarises a whole set of observations. In this case, the error can often be expressed as the sum of the errors that the function produces for each observation separately (as in our example). There are then two main ways that gradient descent can be applied:

- In “batch” mode. That is, we can search for the function that minimises the sum of errors. This means, of course, having all the observations available before we can start.
- Incrementally, i.e. as the observations arrive. When an observation comes in, we perform one interation of gradient descent, moving in a direction that will lessen the error *for that observation only*. When the next one comes, we take one step towards minimising the error for that observation, and so on. Although with a new observation we don’t immediately iterate until the error is minimal for that observation, nevertheless over time (and possibly with repeat presentations of some observations) we can hope to come to a solution that produces small errors on the observations as a whole.

Incremental use of gradient descent is a kind of reinforcement learning. That is each iteration of the gradient descent is adapting the system’s internal representation to be slightly more appropriate for handling that particular observation correctly.

12.5 Background Reading

Gradient descent is discussed in Duda and Hart, p 140.

- Michie, D. and Chambers, R. A., “BOXES: An Experiment in Adaptive Control”, in Dale, E. and Michie, D., Eds., *Machine Intelligence 2*, Oliver and Boyd, 1968.
- Duda, R. and Hart, P., *Pattern Classification and Scene Analysis*, Wiley, 1973.

Chapter 13

Linear Classifiers and the Perceptron

In this lecture we apply the idea of gradient descent in a particular approach to concept learning. This gives rise to a whole class of connectionist learning procedures.

13.1 Linear classification

The set of points in a given class is a subspace of the whole space of possible observations. Linear classification involves trying to find a *hyperplane* that separates the points in the class from everything else[1]. A measure of the the likelihood that an observation belongs to the class can then be obtained by seeing which side of the hyperplane the observation lies on and how far from the hyperplane it is.

Mathematically, (in the discriminant function case; the general case is similar) we attempt to find a function g of the following form:

$$g(\mathbf{x}) = \mathbf{a}^t\mathbf{x} + a_0$$

That is,

$$g(\mathbf{x}) = (\sum_{j=1}^{m} x_{ij}a_j) + a_0 \tag{13.1}$$

(where $\mathbf{x}$ is the ith sample and a_j is the jth component of $\mathbf{a}$). This corresponds to finding the projection of $\mathbf{x}$ onto a vector $\mathbf{a}$ which is normal to the chosen hyperplane. If the value of this projection is $-a_0$ then $\mathbf{x}$ lies exactly in the hyperplane. If the projection is larger, then $\mathbf{x}$ is on the side corresponding to the learned concept; if it is less then $\mathbf{x}$ is not considered to be an instance of the concept (it is on the wrong side of the hyperplane).

---

[1] If the overall space of concepts is m-dimensional, a hyperplane is an infinite subspace of this with dimension $m-1$. Thus, for instance, if there are two variables then linear classification attempts to find a line separating the points in the class from everything else; if there are three variables then it is a plane, etc.

In general, a linear discriminant is computed by defining an appropriate error function for the training sample and then solving for the coefficients $\mathbf{a}$ and a_0 by gradient descent. Different ideas about what the error function should be then give rise to a family of different methods (see Duda and Hart for an extensive description).

One way of measuring the error is to say that error only comes from observations that are wrongly classified. For those $\mathbf{x}$ wrongly classified as *not* being instances of the concept, $-g(\mathbf{x})$ gives a measure of how much error there is. For those wrongly classified in the class, $g(\mathbf{x})$ gives a measure of how wrong the system currently is. Thus:

$$E = \sum_{\mathbf{x}\, wrongly\ classified\ out} -g(\mathbf{x}) + \sum_{\mathbf{x}\, wrongly\ classified\ in} g(\mathbf{x})$$

This is called the *perceptron criterion function.* Now for simplicity let us assume that every observation is augmented with one extra component whose value is always 1, and that a_0 is added on the end of the $\mathbf{a}$ vector (the "weight vector"). This is just a device to get the discriminant function to be in the simpler form

$$g(\mathbf{x}) = \mathbf{a}^t\mathbf{x}$$

Then:

$$E = \sum_{\mathbf{x}\, wrongly\ classified\ out} -\mathbf{a}^t\mathbf{x} + \sum_{\mathbf{x}\, wrongly\ classified\ in} \mathbf{a}^t\mathbf{x}$$

For gradient descent, we need to consider how E depends on each component a_i of the $\mathbf{a}$ vector. Looking back at equation 13.1, it follows easily that:

$$\frac{\partial E}{\partial a_j} = \sum_{\mathbf{x}\, wrongly\ classified\ out} -x_{ij} + \sum_{\mathbf{x}\, wrongly\ classified\ in} x_{ij}$$

Putting the error gradients into a vector for the different a_j and substituting into the equation for gradient descent then gives:

$$\mathbf{a}(t+1) = \mathbf{a}(t) + \rho \sum_{\mathbf{x}\, wrongly\ classified\ out} \mathbf{x} - \rho \sum_{\mathbf{x}\, wrongly\ classified\ in} \mathbf{x})$$

This gives a very simple basis for tuning the weight vector - you simply add in the examples that were wrongly classified out and subtract the examples that were wrongly classified in.

13.2 The Perceptron Convergence Procedure

When an incremental version of gradient descent is used, it is possible to make this into a reinforcement learning system. Adjusting the weights after each training example $\mathbf{x}$ gives rise to the following training procedure:

$a_j(t+1) = a_j(t)$ if $\mathbf{x}(t)$ classified correctly
$a_j(t+1) = a_j(t) + \rho x_j(t)$ if $\mathbf{x}(t)$ wrongly classified out
$a_j(t+1) = a_j(t) - \rho x_j(t)$ if $\mathbf{x}(t)$ wrongly classified in

where $\mathbf{x}(t)$ is the input at time t and x_j is its jth component. This is the *perceptron convergence procedure.*

13.3 The Perceptron

The perceptron is a simple processing unit that takes a set of inputs, corresponding to the values of a set of variables, and produces a single output. Associated with each input x_j is a weight a_j that can be adjusted. The output of the perceptron is determined as follows:

$$\begin{aligned} g(\mathbf{x}) &= 1 \text{ if } \mathbf{a}^t\mathbf{x} > 0 \\ &= 0 \text{ if } \mathbf{a}^t\mathbf{x} \leq 0 \end{aligned}$$

It can be seen from this description that a perceptron is simply an implementation of a linear discriminant function. The standard ways of training a perceptron (that is, causing it to adjust its weights in order to produce better input-output behaviour) include the perceptron convergence procedure described above.

In general, perceptron-based learning systems make of multiple perceptrons, each charged with computing some element of the answer.

13.4 Example: Assigning Roles in Sentences

(McClelland and Kawamoto 1986).

13.4.1 The Task

Given a syntactic analysis of a sentence, associating words (with limited semantics) with syntactic roles: subject, verb, object, object of "with", return a representation of the fillers of the semantic roles: agent, patient, instrument, modifier. This is non-trivial:

The window broke.
The door opened.
The man broke the window with the hammer/curtain.

13.4.2 Network

- Input words encoded in terms of a number of binary semantic features.

- One input unit for each pair of (noun or verb) semantic features for each syntactic role (value 0, 0.5 or 1).
- Each input unit connected (with a weight) to each output unit.
- One group of output units for each semantic role. Each group contains units for each possible conjunction of features from the verb and from the filler (with the modifier role, noun features, rather than verb features, are used).
- Semantic features for each semantic role are obtained by summing.
- Training is by the Perceptron Convergence Procedure.

13.4.3 Results

- Performance on the basic task improves with training.
- The system is enable to hypothesise features for missing roles.
- The system can disambiguate ambiguous words.
- Gradations of meaning.

13.5 Limitations of Perceptrons

The perceptron is probably the simplest in the array of techniques available in the class of *connectionist* learning systems. There are many connectionist models in use, all sharing the common idea of carrying out computation via large numbers of simple, interconnected, processing units, rather than by small numbers of complex processors.

Of course, there is no reason to assume that the points corresponding to a concept can be separated off from the rest by means of a hyperplane (where this is the case, the class is called *linearly separable*). The result of the gradient descent can be expected to give a hyperplane that separates off the concept as well as possible, but it may not be completely successful. It is possible to try to separate off a concept by means of other types of surfaces (for instance, hyperquadratic surfaces). This gives rise to quadratic classifiers, etc.

Since a perceptron can only represent a linearly separable concept, this means that there are many concepts that a perceptron cannot learn. In particular, it is not possible to construct a perceptron that takes two binary inputs and which fires only if exactly one of the inputs is 1 (this is called the **XOR problem**). These kinds of limitations meant that for many years researchers turned away from connectionist models of learning.

The main ingredients of a solution to the problem of the limitations of perceptrons are the following:

- Arranging perceptrons in layers, with some units (*hidden units*) not connected directly to inputs or outputs.
- Replacing the step function by a continuous and differentiable function.

Hidden units give the network "space" to develop its own distributed representations. Having an activation function that can be differentiated means that it is possible to reason about how the error depends on network weights that appear further back than the output layer.

13.6 Some Reflections on Connectionist Learning

- The schemes we have seen are all basically for reinforcement learners using gradient descent in different error functions (note the discussion of Michie's work on p10-11 of Beale and Jackson).
- They will therefore suffer from the same problems that all gradient descent methods have (local minima, divergence).
- The weights of a network can be thought of as the coefficients of a complex equation. Learning is a process of (multi-dimensional) "curve fitting" - finding the best values for these coefficients.
- The numerical nature of the networks allows for elegant solutions to the credit assignment problem.
- Understanding the representations developed by a connectionist learner is very difficult.
- Encoding complex inputs in such a way as to be suitable for input to a connectionist machine can be complex.
- Deciding on an architecture for a given learning problem is more an art than a science.
- Connectionist models cannot directly:
  - Handle inputs of arbitrary length.
  - Represent recursive structure.

13.7 Background Reading

For classifiers, discriminant functions and linear classifiers, see Duda and Hart. Linear classifiers are discussed in Beale and Jackson, section 2.7. Chapters 3 and 4 are also relevant to this lecture, though they go into more detail than is necessary. If you want to find out more about the various types of connectionist learning systems, then you should go to the Connectionist Computing module.

- Duda, R. and Hart, P., *Pattern Classification and Scene Analysis*, Wiley, 1973.
- Beale, R. and Jackson, T., *Neural Computing: An Introduction*, IOP Publishing, 1991.
- McClelland, J. and Kawamoto, A., "Mechanisms of Sentence Processing: Assigning Roles to Constituents of Sentences", in McClelland, J., Rumelhart, D. et al., *Parallel Distributed Processing*, MIT Press, 1986.

Chapter 14

Explanation Based Generalisation (EBG)

Also known as Explanation Based Learning (EBL)

14.1 Demonstration: Finger

The FINGER program is a system that attempts to combine simple actions into "chunks" that will be useful for complex tasks. Given an initial state and a goal state, it searches through possible sequences of the actions it knows to see whether any will transform the initial state into the goal state. This search has to be cut off at some point, which means that some possible solutions will elude it. However, successful actions from the past get added to the basic list of actions available, which means that complex actions involving this as a part will be found more quickly. As a result, careful teaching will enable FINGER to do complex tasks which were initially outside its capabilities.

The FINGER program is based on an idea of Oliver Selfridge and was implemented by Aaron Sloman and Jon Cunningham at the University of Sussex. The sequence of examples given to FINGER is vitally important.

14.2 Learning as Optimisation

- No learning takes place within a complete vacuum.
- The more knowledge is initially available, the more learning is *reformulation*.
- Examples: Chunking, Finger, efficiency improvements, dynamic optimisation during problem solving.

14.3 Explanation Based Learning/ Generalisation

- Knowledge, not data, intensive.
- Guided by *one* example only.
- Proceeds in two steps:
    1. Determining why this is an example (the *explanation*).
    2. Determining how this can be made to cover more cases (the *generalisation*).

14.4 Operationality

Not just any explanation will do – it must be expressed in terms of *operational* concepts. The notion of operationality is domain-dependent – it may correspond to “cheap to use”, “no search/ inference needed”, etc.

14.5 Definition of EBL

14.5.1 Inputs

- Target concept definition (not operational).
- One training example.
- Domain theory (to be used in building the explanation).
- Operationality criterion.

14.5.2 Output

A generalisation of the training example that is a *sufficient* description for the target concept and which is operational. In terms of subsumption,

Example $\leq$ Output $\leq$ Target

14.6 A Logic Interpretation

14.6.1 Explanation

Construct a proof P of the example being an example, using the domain knowledge:

DomainK, ExampleK $\vdash_P$ example(Example)

14.6.2 Generalisation

Determine the minimal information about the example sufficient to let P go through:

DomainK, PartOfExampleK $\vdash_P$ example(Example)

14.6.3 Result

The concept of all things described by this PartOfExampleK.

14.7 The generalisation process (Regression)

Once the proof is obtained, this is generalised by regressing (back-propagating) the target concept (the general one) through the explanation structure. In the Prolog proof case, the "explanation structure" is the sequence of clauses chosen. So regression means carrying out a proof with the same "shape" (the same clauses are chosen in the same sequence) with the target (less instantiated) concept instead of the example.

14.8 Prolog Code for EBL

The following code constructs the generalised proof at the same time as the concrete one:

- Information in the concrete proof (first argument) chooses the clauses.
- The generalised proof (second argument) shadows this.
- The result (third argument) is the leaves of the generalised proof.

```
ebg(Leaf,GenLeaf,GenLeaf) :- operational(Leaf), !, Leaf.
ebg((Goal1,Goal2),(GenGoal1,GenGoal2),(Leaves1,Leaves2)) :- !,
   ebg(Goal1,GenGoal1,Leaves1),
   ebg(Goal2,GenGoal2,Leaves2).
ebg(Goal,GenGoal,Leaves) :-
   clause(GenGoal,GenBody),
   copy((GenGoal :- GenBody),(Goal :- Body)),
   ebg(Body,GenBody,Leaves).
```

14.9 EBG = Partial Evaluation

See (van Harmelen and Bundy 1988).

EBG	PE
Target Concept	Program to be Evaluated
Domain Theory	Program definitions
Operationality	When to stop the execution
(Nothing corresponds)	Partial information about arguments
Guidance by example	(Nothing corresponds)
Result implies Target	Result equivalent to Target (with these arguments)
One guided solution	All solutions

14.10 Reading

- Van Harmelen, F. and Bundy, A., “Explanation-Based Generalisation = Partial Evaluation”, *Artificial Intelligence* Vol 36, pp401-412, 1988.
- Kedar-Cabelli, S. and McCarty, L. T., “Explanation Based Generalisation as Resolution Theorem Proving”, *Procs of the Fourth International Machine Learning Workshop*, Irvine, Ca., 1987.

Chapter 15

Examples of EBL in Practice

15.1 STRIPS MACROPS

This was possibly the first use of EBL techniques, though happened before the notion of EBL was properly formulated.

STRIPS (Fikes et al 1972) was a robot planner, making use of operators of the following kind:

```
OPERATOR: gothru(D1,R1,R2)
PRECONDITIONS: inroom(robot,R1), connects(D1,R1,R2)
ADDS: inroom(robot,R2)
DELETES: inroom(robot,R1)
```

A *triangle table* (Figure 15.1) is a representation for complete plans which have been successful, which facilitates the process of learning new "macro operators". The basic principles for its construction are:

- Row 1 is a single box containing the facts that were initially true in the world.
- Row i ($i > 1$) is a set of boxes containing the facts that were true in the world after the $i - 1$th operator in a plan was executed.

`*inroom(robot,r1)` `*connects(d1,r1,r2)`	`gothru(d1,r1,r2)`	
`*inroom(box1,r2)` `*connects(d1,r1,r2)`	`*inroom(robot,r2)`	`pushthru(box1,d1,r2,r1)`
		`inroom(robot,r1)` `inroom(box1,r1)`

Figure 15.1: A Triangle Table

*inroom(_,_) *connects(_,_,_)	gothru(_,_,_)	
*inroom(_,_) *connects(_,_,_)	*inroom(_,_)	pushthru(_,_,_,_)
		inroom(_,_) inroom(_,_)

Figure 15.2: The Triangle Table Generalised

- Column 0 (the first column) after the first row records those facts from the initial state that were required to be true by the appropriate operator.
- Column i ($i > 0$) tracks the facts added by an operator and how long they last.
- Facts in a row are marked (with a "*") if they are preconditions of the next operator to be executed.

The sequence of operators OP_i, OP_{i+1}, OP_n is a possible "chunk" that can be executed if all the marked facts in the ith "kernel" are true. The ith kernel is the square occupying rows $i+1$ to $n+1$ and columns 0 to $i-1$. MACROPS are formed in the following way:

1. A triangle table is constructed from the plan (the "explanation").
2. This is "generalised" so that its kernels can be used as the preconditions for generalised sequences of operators.

Generalisation has the following stages:

1. Constants are replaced by variables (see Figure 15.2).
2. The recorded operator sequence is "replayed" (i.e. the preconditions are sought within the table and the adds and deletes are matched against entries in the table. This appropriately instantiates the variables. The result for our example table is shown in Figure 15.3.
3. Various other optimisations are performed.

Thus the system has learned the new operator:

```
OPERATOR: gothru(P3,P2,P5) THEN pushthru(P6,P8,P5,P9)
PRECONDITIONS: inroom(robot,P2), connects(P3,P2,P5),
               inroom(P6,P5), connects(P8,P9,P5)
ADDS: inroom(robot,P9), inroom(P6,P9)
DELETES: inroom(robot,P2), inroom(P6,P5)
```

*inroom(robot,P2) *connects(P3,P2,P5)	gothru(P3,P2,P5)	
*inroom(P6,P5) *connects(P8,P9,P5)	*inroom(robot,P5)	pushthru(P6,P8,P5,P9)
		inroom(robot,P9) inroom(P6,P9)

Figure 15.3: Final Triangle Table

15.2 Evaluation of EBL

(Minton 1985) evaluated two versions of a system to acquire MACROPS in a Strips-like scenario. The system MAX, which acquires all possible MACROPS, seems to search a larger search space than a system with no learning, and is comparable in terms of CPU time. The other system, MORRIS, is more selective about saving only those MACROPS which are either frequently used or which represent "non-obvious" steps in terms of the heuristic evaluation function used. Its performance is significantly better. In his paper in *Readings in Machine Learning*, Minton analyses the efficiency effects of EBL in general into three types:

++ **Reordering Effect.** Macro operators, or their equivalent, allow a certain amount of "look-ahead", which can help in the avoidance of local maxima.

+ **Decreased Path Cost.** Macro operators have already precomputed the results of combining sequences of operators. If one of these is chosen, this computation does not have to be done again.

– **Increased Redundancy.** Adding macro operators increases the size of the search space (since the original operators, which in principle could re-derive the macro operators again, are still all there).

Since not all of these are positive effects, it is necessary to be selective about what is learned and remembered.

15.3 LEX2 - Learning Symbolic Integration

The relevant part of LEX (Michell et al 1984) here is the one that keeps track of when particular operators (e.g. integration by parts) should be applied, given positive and negative examples. Version spaces are used for this. Thus, for the operator:

$$op1 : \int r.f(x)dx = r \int f(x)dx$$

if it has applied successfully to $\int 7x^2 dx$ then the version space will be represented by:

G = $\{\int rf(x)dx\}$
S = $\{\int 7x^2 dx\}$

EBL can improve on this (the S value).

In the following example, we consider a use of the above operator followed by a use of the operator:

$$op9 : \int x^{r\neq -1} dx = \frac{x^{r+1}}{(r+1)}$$

to solve the problem $\int 7x^2 dx$.

The steps are as follows:

1. Solve the problem for the example.
2. Produce an explanation of why the example is a "positive instance" for the first operator in the sequence which echoes the trace of the successful solution but which leaves the expression as a variable. This essentially involves constructing here a Prolog-like proof of `pos_inst(op1,`$\int 7x^2 dx$`)` using clauses like:

```
pos_inst(Op,State) :-
   \+ goal(State),
   (goal(apply(Op,State)); solvable(apply(Op,State))).

solvable(State) :-
   goal(apply(_,State)); solvable(apply(Op,State)).
```

   That is, for the operator to be applicable, the state (expression) it is applied to must not be a goal state, and when the operator is applied to that state, the result must either be a goal state or (recursively) solvable. For our example, this gives:

```
pos_inst(op1,State) :-
   \+ goal(State),
   goal(apply(op9,apply(op1,State))).
```

3. Restate this in terms of the generalisation language (i.e. the language of generalisations that gives rise to the description space of possible expressions). In our example,

```
 pos_inst(op1,State) :-
   match(∫ f(x)dx, State),
   match(f(x), apply(op9,apply(op1,State))).
```

That is, the operator applies if the expression is indeed an integral and if the result of applying op9 to the result of applying op1 to it is a non-integral (here $f(x)$ indicates any function of x that is not an integral).

4. Propagate restrictions on operator applicability backwards through the proof. Here the restrictions on op9 reduce the last goal to:

   `match(`$r \int x^{r \neq -1} dx$`, apply(op1,State)).`

   (the two rs denoting possibly different constants), and the restrictions on op1 reduce this to:

   `match(`$\int r x^{r \neq -1} dx$`, State).`

5. Use the new restrictions to generalise the S set for the first operator. Here we would have:

   $$\mathrm{S} = \{\int r x^{r \neq -1} dx\}$$

15.4 SOAR - A General Architecture for Intelligent Problem Solving

SOAR (Laird et al 1986) is a problem solving system based on human memory and performance.

- Problem-solving is a goal-directed search within a problem space which makes available various operators.
- Production systems are used to represent all knowledge. Knowledge is "elaborated" by running rules until the system is quiescent. This knowledge will be used to make decisions.
- When a decision cannot be made, a subgoal is generated to resolve the impasse. The whole problem solving machinery is brought to bear on this. When the subgoal succeeds, execution continues from where it left off.
- Productions are automatically acquired that summarise the processing in a subgoal – "chunking". This process is similar to EBL.

SOAR is interesting because it represents a general problem-solving machine that has learning as a critical part. Moreover there are arguments for its psychological validity.

15.5 Using EBL to Improve a Parser

Samuelsson and Rayner used EBL to improve the performance of a natural language parser. They started off with a general purpose parser for English (the Core Language Engine system developed at SRI, Cambridge) and a set of 1663 sentences from users interacting with a particular database, all of which the parser could handle. 1563 sentences were used as examples to derive new rules by EBL. The parser was written as a DCG, and so EBL for this was a straightforward application of EBL for Prolog programs. The result was 680 learned rules. With these rules, the parser could cover about 90% of the remaining 100 sentences from the corpus. Using the learned rules first, and only using the main grammar if they failed, led to a speedup of parser performance of roughly 3 times.

15.6 References

Thornton Chapter 8 discusses EBL and LEX.

- Fikes, R. E., Hart, P. E. and Nilsson, N. J., "Learning and Executing Generalised Robot Plans", *Artificial Intelligence* Vol 3, pp251-288, 1972.
- Minton, S., "Selectively Generalising Plans for Problem Solving", Procs of IJCAI-85.
- Mitchell, T. M., Utgoff, P. E. and Banerji, R., "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics", in Michalski, R., Carbonell, J. and Mitchell, T., Eds., *Machine Learning: An Artificial Intelligence Approach*, Springer Verlag, 1984 (especially section 6.4.2).
- Laird, J. E., Rosenbloom, P. S. and Newell, A., "Chunking in SOAR: The Anatomy of a General Learning Mechanism", *Machine Learning* Vol 1, pp11-46, 1986.
- Samuelsson, C. and Rayner, M., "Quantitative Evaluation of Explanation-Based Learning as an Optimization Tool for Large-Scale Natural Language System", *Procs of 12th IJCAI*, Sydney, Australia, 1991.

Chapter 16

Unsupervised Learning

In some sense, learning is just reorganising some input knowledge (e.g. finding a more compact way of representing a set of examples and non-examples). Indeed, unsupervised learning is to do with finding useful patterns and generalisations from data in a way that is not mediated by a teacher. These amount to alternative ways of reorganising the data. But what reorganisations are best? Here are some ideas about what one might attempt to optimise:

- The "compactness" of the representations.
- The "informativeness" of representations – their usefulness in minimising "uncertainty".

In this lecture, we will see another possible criterion, Category Utility.

16.1 Mathematical approaches to Unsupervised Learning

In unsupervised learning, the learner is presented with a set of observations and given little guidance about what to look for. In some sense, the learner is supposed to find useful characterisations of the data. Unsupervised learning is supposed to entail the learner being given minimal information about what form the answer might take, though it is hard to define a strict boundary between supervised and unsupervised. The following techniques are ways of finding patterns in the data which assume very little beyond ideas like "it is useful to look at similarities between elements of the same class".

16.2 Clustering

Clustering, or cluster analysis, involves finding the groups of observations that are most similar to one another. It can be useful for a human observer to have groups of similar observations pointed out, because these may correspond to new and useful concepts that have not previously been articulated. Similarly, clustering can be a useful first step for an unsupervised learner trying to make sense of the world.

Cluster analysis may generate a hierarchy of groups - this is called *hierarchical cluster analysis*. The results of a cluster analysis are commonly displayed in the form of a *dendrogram* showing the hierarchy of groups and the degree of similarity within a group.

Cluster analysis can be achieved by *divisive clustering*, where the system starts off with all points in the same cluster, finds a way of dividing this cluster and then subdivides the resulting subclusters in the same way. In practice, this is used less often than *agglomerative clustering*, which constructs a set of clusters D in the following way:

1. Set D to the set of singleton sets such that each set contains a unique observation.
2. Until D only has one element, do the following:
   (a) For each pair of elements of D, work out a similarity measure between them (based on the inverse of a distance metric).
   (b) Take the two elements of D that are most similar and merge them into a single element of D (remembering for later how this element was built up).

The only thing to do now is define how the similarity between two clusters is measured. This involves first of all picking a distance metric for individual observations. This is then extended to a measure of distance between clusters in one of the following ways:

- Single-linkage (or nearest neighbour). The distance between two clusters is the distance between their closest points.
- Complete-linkage (or furthest neighbour). The distance between two clusters is the distance between their most distant points.
- Centroid method. The distance between two clusters is the distance between their means.
- Group-average method. The distance between two clusters is the average of the distances for all pairs of points, one from each cluster.

The similarity between two clusters is then a quantity that behaves inversely to the computed distance (e.g., if d is the distance, $-d$ or $1/d$).

There are many algorithms for cluster analysis but unfortunately no generally accepted 'best' way.

(Finch and Chater 1991) use cluster analysis in the induction of linguistic categories. They start with a 33 million word corpus of English and collect for each of 1000 "focus" words the number of times that each of 150 "context" words occurs immediately before it, two words before it, immediately after it and two words after it. Thus each focus word is associated with a vector whose length is 4 x the number of context words. A statistical distance measure is used between these vectors and used as the basis of a hierarchical cluster analysis. This reveals very clearly categories that we would label as "verb" and "noun" (with some complications) and a more detailed analysis that, for instance, records *women* as closest to *man* and closely related to *people* and *americans*.

16.3 Principal components analysis

Principal components analysis is a simple kind of "change of representation" that can give a more revealing view of a set of observations than the original set. The choice of variables to measure in a learning situation is not always obvious, and principal components analysis suggests an alternative set, derived from the original ones, which are uncorrelated with one another (that is, the covariances between different variables are zero). The idea is to derive a set of independent dimensions along which the observations vary, using combinations of the original (often dependent) variables.

For instance, one might decide to measure various aspects of a set of birds, including the beak size, age and total length. Unfortunately, beak size and total length are probably both related to the overall size of the bird and hence correlated to one another. Principal components analysis might propose using a variable like 0.4 x beak size + 0.9 x total length (a measure of overall size) instead of the separate beak size and total length variables.

It turns out (see Appendix A.1) that the vectors expressing the principal components in terms of the original set of variables are the *eigenvectors* of $\mathbf{C}$, and the variances of the new components, λ_i, the *eigenvalues*. There are standard numerical techniques for computing these. Thus it is very straightforward to calculate the principal components by standardising the variables, calculating the covariance matrix and then finding its eigenvectors and -values.

From the eigenvalues λ_i it is possible to see from these how much of the variation of the original observations is accounted for by each of the components. For instance, if the eigenvalues were 2, 1.1, 0.8, 0.1, (summing to 4), then the first principal component accounts for $(100*2/4) = 50\%$ of the variation observed in the population, whereas the last one only accounts for $(100*0.1/4) = 2.5\%$. If one of the eigenvalues is zero, it means that there is no variation in the value of that component, and so that component can be missed out in describing observations. Even if a variance is very small but not zero, this may be good evidence for ignoring that component.

16.4 Problems with conventional clustering

Conventional clustering is weak in the following respects:

- It assumes a “batch” model (is not incremental).
- The resulting “concepts” may be incomprehensible/ very complex.
- There is no way to have knowledge/ context affect the classification.

16.5 Conceptual Clustering

The clustering problem: Given:

- A set of objects.
- A set of measured attributes.
- Knowledge of:
  - problem constraints
  - properties of attributes
  - a goodness criterion for classifications

Produce:

- A hierarchy of object classes:
  - each class described by a concept (preferably conjunctive)
  - subclasses of a parent logically disjoint
  - goodness criterion optimised

Conceptual clustering builds a hierarchy of object classes, in the form of a discrimination tree, in an incremental manner.

16.6 UNIMEM

In UNIMEM, the tree is made out of nodes storing the following information:

- A set of instances.
- A set of shared properties.

A new object (instance) is incorporated into the tree according to the following algorithm (the following is a simplified description of the real thing):

1. Find the most specific nodes whose shared properties describe the instance.
2. For each one, add it to the set of instances.
3. Where two instances at a node have "enough" properties in common, create a child node with these instances and their shared properties.

UNIMEM is claimed to be an approach to "generalisation based memory" – this method of storing instances enhances the retrieval of information (by inheritance). It is more similar to *divisive* than *agglomerative* clustering. The method is incremental and produces comprehensible conjunctive concepts. However, the system has many different parameters which contribute to its notion of "goodness" of the taxonomy.

16.7 COBWEB

16.7.1 Category Utility

What makes a good classification scheme? Fisher based his COBWEB system on an explicit criterion based on the results of psychological work on "basic categories". Ideally, one wants to maximise two quantities:

Intra-Class Similarity. The ability to *predict* things from class membership. Formally, $P(property|class)$.

Inter-Class Dissimilarity. The ability to predict the class from the properties of an instance. Formally, $P(class|property)$.

One way of combining these two into an evaluation function would be to compute

$$\sum_{class\ c} \sum_{property\ p} P(p).P(c|p).P(p|c)$$

which, by Bayes rule, is the same as:

$$\sum_{class\ c} P(c). \sum_{property\ p} P(p|c)^2$$

Fisher defined *category utility* (CU) as the *increase* in this compared to when there is just one category, divided by the number of categories:

$$CU(\{c_1, c_2 \ldots c_n\}) = (1/n) \sum_{i=1}^{n} P(c_i)[\sum_{p} P(p|c_i)^2 - \sum_{p} P(p)^2]$$

16.7.2 The Algorithm

A node in COBWEB has the following information:

- The number of instances under that node.
- For each property p, the number of these instances that have p.

The following recursive algorithm adds example (instance) E to a tree with the root node R (this has been simplified slightly).

1. Increment the counts in R to take account of the new instance E.
2. If R is a leaf node, add a copy of the old R and E as children of R.
3. If R is not a leaf node,
    (a) Evaluate the CU of adding E as a new child of R.
    (b) For each existing child of R, evaluate the CU of combining E with that child.

    According to which is best:

    (a) Add E as a new child to R, OR
    (b) Recursively add E to the tree whose root is the best child.

16.7.3 Comments on COBWEB

COBWEB makes use of a very elegant notion of what is a “good classification”. But the kind of hill-climbing search through possibilities that it carries out will mean that the results depend very much on the order of presentation of the data (the instances). The complete COBWEB algorithm has extra operations of node merging and splitting, but still the idea is the same - at each point, choose the possibility that gives the greatest CU score.

Learning as maximising category utility - a kind of optimisation.

16.8 Unsupervised Learning and Information

Finally we consider what the relationship is between unsupervised learning and information theory. Unsupervised learning can be taken to be the task of translating the input data into output data in the most informative way possible. That is, the entropy of the final situation should be minimised. But this would be achieved by encoding everything as the same output!

We also need to minimise the “ambiguity” of the output – the *equivocation* (or the entropy of the input when the output is known). If the output is a good way of representing the important features of the input then it should be possible to reconstruct much of it from the output. The equivocation is given by the formula:

$$-\sum_{\mathrm{input}\,i,\mathrm{output}\,o} P(i \wedge o)\log_2(P(i|o))$$

There is a tradeoff here. We need to have the output reflect as much as possible of the input. This will be achieved if it captures genuine generalisations/ similarities. Unfortunately this does not in itself tell us how to go about finding such a good encoding of the input.

16.9 References

Conventional clustering is described in Manly Chapter 8. Principal components analysis is described in Manly Chapter 5.

Thornton Chapter 7 is an introduction to the idea of clustering, though it spends more time on UNIMEM and less on COBWEB than we do.

- Finch, S. and Chater, N., “A Hybrid Approach to the Automatic Learning of Linguistic Categories”, *AISB Quarterly* No 78, 1991.
- Manly, B. F. J., *Multivariate Statistical Methods*, Chapman and Hall, 1986.
- Fisher, D. H., “Knowledge Acquisition via Incremental Conceptual Clustering”, *Machine Learning* Vol 2, 1987 (also in *Readings in Machine Learning*).
- Ehrenberg, A. S. C., *A Primer in Data Reduction*, Wiley, 1982.

Chapter 17

Knowledge Rich Learning - AM

Up to now, the learning systems that we have considered have had access to hardly any knowledge of the world (basically just the shape of the underlying description space). In this lecture, we consider an extreme case of knowledge-aided learning, Lenat's AM (Automated Mathematician) system. AM is an example of an unsupervised learning system that is let loose to discover "interesting things" in a domain. It is guided by a great deal of knowledge about how to go about that task.

17.1 Mathematical Discovery as Search

Mathematical discovery can be viewed as a search process. At any point in time, a particular set of mathematical concepts are known and accepted. These concepts have known examples and relationships to one another. Making a scientific breakthrough involves coming up with a new concept that turns out to be very interesting, or making a conjecture about how two concepts are related. But such new concepts and conjectures are not usually unrelated to what was known before. Lenat hypothesised that there are heuristic rules that one can use to derive new concepts and conjectures from existing ones.

17.2 The Architecture of AM

17.2.1 Representation of Concepts

AM represents concepts as frames with slots or "facets". Most of the activity of the system is to do with filling these facets. For mathematical concepts, the facets used include:

Name. May be provided by the user or created by the system.

Generalisations. Concepts that are more general than this one.

Specialisations. Concepts that are more specific than this one.

Examples. Individuals that satisfy this concept's definition.

In-domain-of. Operations that can act on instances of the concept.

In-range-of. Operations that produce instances of the concept.

Views. Ways of viewing other objects as instances of this concept.

Intuitions. Mappings from the concept to some standard scenario.

Analogies. Similar concepts.

Conjectures. Potential theorems involving the concept.

Definitions. May include LISP code for determining whether something is an instance or not.

Algorithms. Appropriate if the concept is a kind of operation.

Domain/Range. ditto.

Worth. How useful/ valuable is the concept?

Interestingness. What features make the concept interesting?

Associated with a facet F are subfacets, as follows:

F.Fillin. Methods for filling in the facet.

F.Check. Methods for checking/ fixing potential entries.

F.Suggest. New tasks relevant to the facet that might be worth doing if AM "bogs down".

17.2.2 The Agenda

At any point in time, there are many possible things to do. The agenda is used to impose a best-first strategy on this search through possible concepts and facet-fillings. Each entry on the agenda is a task to fill in some facet for some concept. Each task is given a priority rating and the highest priority task is chosen at each point. The priority rating is also used to limit the amount of time and space that the computation is allowed to consume before another task is chosen.

17.2.3 The Heuristics

Once a task is selected, AM selects (using inheritance) heuristic rules attached to the chosen facet. A rule has a conjunctive condition and a set of actions, each of which is a LISP function that can only achieve the following side-effects:

- Adding a new task to the agenda.
- Dictating how a new concept is to be defined.
- Adding an entry to some facet of some concept.

Here are some examples of heuristic rules (translated into English):

> IF the current task is "fill in examples of X"
>   and concept X is a predicate
>   and over 100 items are known in the domain of X
>   and at least 10 cpu secs have been spent so far
>   and X has returned True at least once
>   and X returned False over 20 times as often as True
> THEN add the following task to the agenda:
>   "Fill in the generalisations of X"
>   for the following reasons:
>   "X is rarely satisfied; a slightly more restrictive concept might be
>   more interesting"
>   This reason has a rating with is the False/True ratio.
>
> IF the current task is "Fill in examples of F"
>   and F is an operation, with domain A and range B
>   and more than 100 items are known examples of A
>   and more than 10 range items were found by applying F to these
>   elements
>   and at least one of these range items b is a distinguished
>   member (especially, an extremum) of B
> THEN for each such b in B, create the following concept:
>   NAME: F-inverse-of-b
>   DEFINITION: lambda a. F(a) is a b
>   GENERALISATIONS: A
>   WORTH: average(worth(A),worth(B),worth(b),||examples(B)||)
>   INTEREST: Any conjecture involving both this concept and either
>   F or inverse(F)
>   and . . .
>
> IF the current task is "Fill in examples of F"
>   and F is an operation with domain D

and there is a fast known algorithm for F
THEN one way to get examples of F is to run F's algorithm on
randomly selected examples of D.

17.3 Types of Knowledge given to AM

Knowledge of many forms is provided to AM: the initial concepts and their facet values; the heuristic rules; the evaluation functions; special-purpose LISP code for many functions. AM is exceedingly ambitious in attempting to address complex problems like analogy and the analysis of algorithms. Lenat admits that necessarily some of the solutions were very special-purpose.

17.4 Performance of AM

AM started off with about 30 concepts from finite set theory and 242 heuristic rules attached to various places in the knowledge base. It "discovered" most of the obvious set-theoretic relations (e.g. de Morgan's laws), though these were phrased rather obscurely. After a while, it decided that "equality" was worth generalising, and it came up with the concept of "same size as" and hence natural numbers. Addition was discovered as an analogue of set union and multiplication as a repeated substitution (multiplication was also rediscovered in several other ways). The connection "N+N = 2*N" was discovered. Inverting multiplication gave rise to the notion of "divisors of". Specialising the range of this function to doubletons then gave rise to the concept of prime numbers. AM conjectured the fundamental theorem of arithmetic (unique factorisation) and Goldbach's conjecture (every even number greater than 2 is the sum of two primes). AM also discovered some concepts that are not generally known, such as the concept of maximally divisible numbers.

In a run starting with 115 concepts, AM developed 185 more concepts, of which 25 were "winners", 100 acceptable and 60 "losers". This seems to indicate that the heuristics are doing a good job at focussing the exploration on good directions, and that the space of good concepts is fairly "dense" around the set of starting concepts.

The performance of AM looks impressive, but AM is a very complex system and the published accounts do not always give a consistent picture of exactly how it worked (Ritchie and Hanna 1984). Clearly with such a complex system some simplification is needed for its presentation, though in some cases Lenat seems to have given a misleadingly simple picture of the system's workings. It is not completely clear, for instance, to what extent the heuristic rules have a clear restricted form and to what extent arbitrary LISP code appears. There seems to be little doubt that the system did indeed achieve what is claimed, but the problem is deciding whether this really was a consequence of the simple and elegant architecture that Lenat sometimes describes.

17.5 Conclusions

Knowledge-rich learning is very hard to evaluate, because there is a fine line between giving a system comprehensive background knowledge and predisposing the system to achieve some desired goal. In practice, as with AM, opinions may differ on how significant a given learning system is.

A system like AM is simply too complex to easily evaluate. We shall therefore move on to consider knowledge-based learning frameworks where the knowledge to be used is much more constrained.

17.6 Reading

- Lenat, D. B., “Automated Theory Formation in Mathematics”, Procs of IJCAI-5, 1977.
- Handbook of AI, pp 438-451.
- Lenat, D. B., “AM: Discovery in Mathematics as Heuristic Search”, in Davis, R. and Lenat, D. B., *Knowledge Based Systems in Artificial Intelligence*, McGraw-Hill, 1982.
- Ritchie, G. and Hanna, F., “AM: A Case Study in AI Methodology”, *Artificial Intelligence* Vol 23, pp249-268, 1984.

Chapter 18

Theoretical Perspectives on Learning

In this chapter, we stand back a bit from particular approaches to learning and consider again the problem of what learning *is* and when we can guarantee that it is achieved. We present two definitions of learning that have been proposed. These have spawned a great deal of theoretical work investigating what is, and what is not, learnable. Unfortunately, at present there is still a significant gap between the results of the theorists and the results of practical experience. Reducing this gap is an important goal for future research.

18.1 Gold - Identifiability in the Limit

Gold (1967) considers the problem of language learning, but his approach can be taken to apply to concept learning more generally. I will here express Gold's ideas in this more general setting. It has the following elements:

- The learning process occurs in discrete steps. At each time t, the learner is given some piece of information i_t about the concept.
- Having received the latest piece of information, the learner constructs a guess for the concept, which may depend on all the pieces of‘ data received up to that point:
$$g_t = G(i_1, i_2, ...i_t)$$

The concept C is said to be *identified in the limit* if after some finite amount of time all the guesses are equivalent to C. Thus the learner is allowed some initial confusion, but in order to be said to have learned it must eventually come down to a single correct answer.

Consider a class of concepts, for instance, the class of concepts that can be represented by finite formulae in some logic. That class is called *identifiable in the limit* if there is an algorithm for making guesses that has the following property:

> Given any concept C in the class and any allowable training sequence for the concept (i.e. any allowable sequence of i_ts), the concept C will be identified in the limit.

For Gold, the interesting concepts are the possible languages. The classes of concepts are classes such as the context-free and the context-sensitive languages. Gold considers two methods of *information presentation* -

1. at time t, i_t is an example string of the language (and every string will eventually appear). This is called information via a *text*.
2. at time t, i_t is a yes/no answer to a question posed by the learner itself, as to whether some string is in the language or not. This is called information via an *informant*.

Gold shows that if information is provided via text then the class of finite cardinality languages is identifiable in the limit, but most other classes (regular, context-free, context-sensitive) are not. If information is provided by an informant then language classes up to the class of primitive recursive languages are identifiable in the limit.

18.2 Valiant - PAC Learning

Gold's criterion for learnability has some good features, but learnability requires that the concept be identified for *all* training sequences, no matter how unrepresentative. This means that the number of steps required by a system guaranteed to achieve Gold-learnability will be much greater than the number needed if the examples are random or specially selected. So systems designed to achieve this kind of learnability will not necessarily be very interesting in practice.

Valiant's (1984) model of learnability takes into account the fact that samples of information from the world have some statistical distribution, and it requires that samples are randomly generated. Because there is a small probability that the random samples will give an inaccurate picture of the concept, the model allows the system to be wrong, but with only a small probability. Buntine (1989) quotes a standard informal version of the model as follows:

> The idea is that after randomly sampling classified examples of a concept, an identification procedure should conjecture a concept that "with high probability" is "not too different" from the correct concept.

The phrase "probably approximately correct" (PAC) has given rise to the term "PAC-learning" for the study of learning systems that meet this criterion. One of the formal definitions of "PAC-learnability" (there are a number of variations on the basic theme) is as follows.

Valiant considers the objects to be learned to be programs. For concept learning, we are interested in programs f such that $f(x)$ is 1 if x is an instance of the concept and 0 otherwise. A program f has a "size" $T(f)$. The learner is allowed access to information about f only through a function $EXAMPLES(f)$, which can be called to produce either a positive or a negative example. When it is called, with POS or NEG indicated, the example returned is randomly generated according to some fixed (but unknown) probability distributions D^+ and D^-. These distributions just have to satisfy the obvious restrictions:

$$\begin{aligned}
&\textstyle\sum_{f(\mathbf{x})=0} D^-(\mathbf{x}) = 1\\
&\textstyle\sum_{f(\mathbf{x})=0} D^+(\mathbf{x}) = 0\\
&\textstyle\sum_{f(\mathbf{x})=1} D^-(\mathbf{x}) = 0\\
&\textstyle\sum_{f(\mathbf{x})=1} D^+(\mathbf{x}) = 1
\end{aligned}$$

If F is some class of programs, then F is *learnable from examples* if there is a polynomial p and a learning algorithm A (which has access to information about the concept being learned only through $EXAMPLES$), such that:

for every f in F,
for every possible D^- and D^+ satisfying the above conditions,
for every $\epsilon > 0$,
for every $\delta > 0$,
  A halts in time $p(T(f), 1/\epsilon, 1/\delta)$, outputting program g in F.
  With probability at least $1-\delta$, $\sum_{g(\mathbf{x})=0} D^+(\mathbf{x}) < \epsilon$.
  With probability at least $1-\delta$, $\sum_{g(\mathbf{x})=1} D^-(\mathbf{x}) < \epsilon$.

18.3 Criticisms of PAC Learning

Variations on the above definition have motivated a great deal of theoretical research into what is, and what is not, learnable. One of the advantages of the framework is that, given some desired ϵ and δ and with H the number of possible concepts, it predicts how many examples will be required to guarantee learnability:

$$\epsilon < \frac{H \log(1/\delta)}{N}$$

However, the framework has been criticised (Buntine 1989) for dramatically overestimating the number of samples that are required for learning in practice. As a result, there is a gap between theory and practice that needs to be bridged. For instance, the PAC definition assumes the worst case (learning has to work with even the worst f) rather than the average case, and it ignores the fact that there are often preferences between hypotheses (we are often looking for the "best" concept that matches the data, in some sense).

18.4 Reading

- Buntine, W., “A Critique of the Valiant Model”, IJCAI-89, pp837-842.
- Gold, E. M., “Language Identification in the Limit”, *Information and Control* 10, pp447-474, 1967.
- Pitt, L. and Valiant, L. G., “Computational Limitations on Learning from Examples”, *JACM* Vol 35, No 4, 1988.
- Valiant, L. G., “A Theory of the Learnable”, *CACM* Vol 27, No 11, 1984.

Appendix A

Appendices

Note that the material in these Appendices is for information only, and is not part of the module materials that you are supposed to know.

A.1 Principal Components and Eigenvectors

The aim of principal components analysis is to derive a new representation for the same data such that the covariance matrix $\mathbf{C}$ is diagonal. Each new variable will be a linear combination of the original ones, i.e. if the new variables are y_i then:

$$y_i = \sum_{j=1}^{m} a_{ji} x_j$$

We can put the values a_{ji} into a matrix $\mathbf{A}$. The columns of $\mathbf{A}$ are then the definitions of the new variables in terms of the old ones.

Changing to the new variables is a transformation of the coordinate system; if $\mathbf{x}$ is the coordinates of an observation using the old variables and $\mathbf{y}$ the coordinates using the new ones then:

$$\mathbf{x} = \mathbf{A}\mathbf{y}$$

There is a question of how we should scale the new variables - if we pick a new variable y_i then clearly $2y_i$ would be just as good as a variable. To standardise, it is usual to assume that

$$\sum_{j=1}^{m} a_{ji}^2 = 1$$

This means that the columns of $\mathbf{A}$ are unit vectors in the directions of the new variables (expressed in terms of the old variables). These unit vectors must be at right angles to one another (otherwise there would be correlations between them). This combination of properties means that $\mathbf{A}$ is an *orthogonal* matrix, i.e.

$$\mathbf{A}^t\mathbf{A} = \mathbf{I}$$

(where $\mathbf{I}$ is the identity matrix), and hence:

$$\mathbf{A}^{-1} = \mathbf{A}^t$$

Now we require that the correlation matrix for the new coordinate system is diagonal. Since the new coordinates for an observation $\mathbf{x}$ are given by $\mathbf{A}^{-1}\mathbf{x}$, by equation 8.1 we require:

$$\frac{1}{(n-1)} \sum_{observations\ \mathbf{x}} \mathbf{A}^{-1}\mathbf{x}(\mathbf{A}^{-1}\mathbf{x})^t = \lambda$$

where λ is diagonal. This assumes that we have standardised the original variables to have means of 0. It is standard practice to do this in principal components analysis, as it is also to standardise the variables so that they have variances of 1 (this is achieved by dividing all values by the square root of the original variance). This procedure avoids one variable having undue influence on the analysis. Thus:

$$\begin{aligned}
\lambda &= \frac{1}{(n-1)} \sum_{observations\ \mathbf{x}} \mathbf{A}^{-1}\mathbf{x}\mathbf{x}^t(\mathbf{A}^{-1})^t && \text{(A.1)}\\
&= \mathbf{A}^{-1}\frac{1}{(n-1)}\Big(\sum_{observations\ \mathbf{x}} \mathbf{x}\mathbf{x}^t\Big)\,(\mathbf{A}^{-1})^t && \text{(A.2)}\\
&= \mathbf{A}^{-1}\mathbf{C}(\mathbf{A}^{-1})^t && \text{(A.3)}\\
&= \mathbf{A}^{-1}\mathbf{C}\mathbf{A} && \text{(A.4)}\\
& && \text{(A.5)}
\end{aligned}$$

(where $\mathbf{C}$ is the original covariance matrix) since $\mathbf{A}$ is orthogonal. Hence:

$$\mathbf{C}\mathbf{A} = \mathbf{A}\lambda$$

and for each column $\mathbf{A}_i$ of $\mathbf{A}$:

$$\mathbf{C}\mathbf{A}_i = \lambda_i\mathbf{A}_i$$

(where λ_i is the ith diagonal element of λ). The vectors $\mathbf{A}_i$ satisfying this equation are called the *eigenvectors* of $\mathbf{C}$, and the values λ_i the *eigenvalues*. There are standard numerical techniques for computing these. Thus it is very straightforward to calculate the principal components by standardising the variables, calculating the covariance matrix and then finding its eigenvectors and -values.

Index

agglomerative clustering 98
AM 105
AQ11 47
AQ 29
batch mode 80
Bayesian classification 60
bias 14
bias 18
bias 78
candidate elimination algorithm 27
case based learning 52
case based reasoning 54
category utility 101
CBR 54
chi-square test 59
CIGOL 42
classifier 49
CLS 68
clustering 97
COBWEB 101
conceptual clustering 100
concept 18
conjunctive descriptions 18
covariance 57
cover 20
cross validation 76
decision tree 48
decision tree 67
dendrogram 98
description space 19
devisive clustering 98
dimension 18
discovery 105
discriminant function 18
EBG 87
EBL 87
Euclidean metric 51
entropy 64
explanation based generalisation 87
explanation based learning 87
exploitation 77
exploration 77
features 17
FOIL 39
gain 79
generalisation operator 20
generalisation 78
Gold 111
gradient descent 78
hierarchical cluster analysis 98
hyperplane 81
ID3 69
identifiability in the limit 111
ILP 31
incremental learning 80
incremental 25
inductive logic programming 31
information theory 63
instance based learning 52
interpolation 78
least squares fitting 79
LEX 93
linear classification 81
linearly separable 84
MACROP 91
Mahalanobis distance 61
Manhatten metric 51
mean 57
MIS 32
multivariate normal distribution 59

nearest neighbour classification 51
nominal value 19
observation 17
operationality 88
PAC learning 112
partial evaluation 90
perceptron convergence procedure 83
perceptron criterion function 82
perceptron 83
population 17
principal components analysis 99
refinement operator 20
refinement operator 34
sample 17
SOAR 95
standard deviation 58
STRIPS 91
structured value 19
triangle table 91
UNIMEM 100
Valiant 112
variable 17
variance 58
version space 27
windowing 70
XOR 84