Predicting the Effect of Instance Representations on Inductive Learning

Sharad Saxena

Department of Computer and Information Science
University of Massachusetts, Amherst, MA 01003.

COINS Technical Report 91-58
August 14, 1991

Predicting the Effect of Instance Representations on Inductive Learning

A Dissertation Presented

by

Sharad Saxena

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

September 1991

Department of Computer and Information Sciences

PREDICTING THE EFFECT OF INSTANCE REPRESENTATIONS ON INDUCTIVE LEARNING

A Dissertation Presented

by

SHARAD SAXENA

Approved as to style and content by:

Paul E. Utgoff, Chair of Committee

Andrew G. Barto, Member

David A. Mix Barrington, Member

Maciej J. Ciesielski, Member

W. Richards Adrion, Department Chair
Computer and Information Science

ACKNOWLEDGEMENTS

This research was supported by the National Science Foundation under the grant number IRI-8619107 and by the Office of Naval Research through a University Research Initiative Program under the contract number N00014-86-K-0764.

I thank Professor Paul Utgoff for all that he has taught me, and for instilling in me the joy of inquiry. His keen insight into the important issues has helped me keep sight of the goal and not get lost in the details. His constant encouragement and support for my ideas, including the ones that did not pan out, gave me the courage to explore the ideas that seemed at first to have only a tenuous connection with the issues I was interested in addressing. I have benefited immensely from his insistence on clear and simple writing.

I thank Professor Andy Barto for his suggestions and comments on the various aspects of this research. His insistence on rigor and thoroughness has taught me to evaluate my ideas critically. I thank Professor David Mix Barrington for imparting to me some of his enthusiasm about the theory of computer science. His ability to make mathematical thought seem effortless has been a source of inspiration. I thank Professor Maciej Ciesielski for careful reading of my dissertation and for providing a fresh viewpoint on my research.

I thank Rick Yee for his friendship and for our many fascinating discussions. His ability to ask the most basic and fundamental questions will always remind me that there is so much more to learn. I also thank Rick for patiently listening to my ideas and helping me express them more clearly and explore them more substantially.

I thank Jamie Callan for going through graduate school with me. He has always been a source of friendly and mature advice. His help, encouragement, and advice

have helped me retain my perspective when things did not seem to be going right. I also thank Jamie for having the time and energy to try ACR in his own research.

I thank Margie Connell for all the help and encouragement, and for the careful reading of my papers. Her comments and suggestions have helped me in presenting my ideas clearly and simply.

I thank Tom Fawcett, Carla Brodly, and Jeff Clouse for making the Machine Learning Lab an interesting and friendly place.

I thank Krishnan, Sandeep, Vijay, Raghu, and Kartik for their friendship. They have made my stay in Amherst happy and enjoyable.

I thank Monica and Parag for all the affection and unconditional support. They provided me with the love and the strength required to survive graduate school.

I thank Ma and Papa for believing in me and for making the sacrifices necessary to give me an opportunity to pursue my dreams.

Finally, I thank Jayashree for going through the joys and sorrows with me. For being there when I needed somebody the most.

ABSTRACT

PREDICTING THE EFFECT OF INSTANCE REPRESENTATIONS ON INDUCTIVE LEARNING

SEPTEMBER 1991

SHARAD SAXENA, B. TECH., INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR, INDIA

M.S., UNIVERSITY OF MASSACHUSETTS

PH.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Paul E. Utgoff

The ability to generalize from examples depends on the algorithm employed for learning and the *instance representation* used to describe the examples to the learning algorithm. Recently a considerable effort has been devoted to the design of algorithms that generalize well in a large number of situations. However, the effect of the instance representation on the accuracy of the generalizations made by the learning algorithm is poorly understood.

This dissertation describes how the duality between finding a compact description for the examples and generalizing from the examples can be utilized to determine the suitability of an instance representation for a learning algorithm. In particular, a heuristic algorithm to compare representations, called ACR, is described. Given a learning algorithm, a set of examples, and alternative instance representations for the examples, ACR attempts to identify the instance representation that will enable the learning algorithm to produce the most accurate hypothesis. For each instance representation, ACR estimates the minimum number of bits with which the learning algorithm can express the examples. Experiments with a variety of learning tasks

show that ACR is effective in ranking representations. In addition, ACR was found to rank representations faster than rankings obtained by directly estimating the accuracy of the hypotheses produced with different representations.

The conclusion that one representation is better than another because it improves the compressibility of the examples suggests that different methods for improving the compressibility of the examples should result in different techniques for improving representations. Two types of representation change are identified by analyzing the reason for change in compresibility of the examples with the different representations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Generalization from examples is central to the ability of machines to learn from experience. Building computational systems that do not require explicit programming, and building systems with the ability to adapt to their changing environment, are among the main goals of machine learning. A system that can learn from examples can acquire the ability to perform a task, or modify its existing procedure for performing a task, by observing a few examples and generalizing from them to unobserved situations.

Two factors that affect the ability to generalize from examples are the algorithm employed for learning and the *instance representation* that describes the examples to the learning algorithm. The design of learning algorithms that generalize well in a variety of domains has received a considerable attention recently. ID3 (Quinlan, 1983), BACKPROP (Rumelhart, Hinton & Williams, 1986), and PT1 (Utgoff, 1989) are representative of these efforts. However, the effect of instance representation on the accuracy of the generalizations made by a learning algorithm is poorly understood.

One of the first decisions that a designer of a learning system must make is to choose a set of features for describing the examples to a learning algorithm. For example, in order to build a system that learns to recognize handwritten characters, many possible representations can be used to describe handwritten characters to a learning algorithm. For instance, like Casey and Nagy (1984) one can digitize each character, embed it in a grid, and for each resulting pixel tell the learning algorithm whether it is on or off. Or, like Arakawa, Odaka and Masuda (1978), one can track the X and Y coordinates of the pen position, compute the Fourier transform of the resulting sequence, and for each character give the Fourier coefficients to the learning algorithm. Or, like a number of other researchers, one can represent the characters by using some other features of the input (Tappert, Suen & Wakahara, 1990). One would like to choose the representation that enables the learning algorithm to produce a classifier that classifies the characters most accurately in the future.

Figure 1.1 shows the difference in the ability of the ID3 learning algorithm (Quinlan, 1986) to generalize with the Pixel and the Fourier representations of handwritten characters. The abscissa gives the number of examples selected randomly and presented to ID3. The examples are selected with uniform probability and replacement from a set of training examples. The ordinate gives the average accuracy obtained

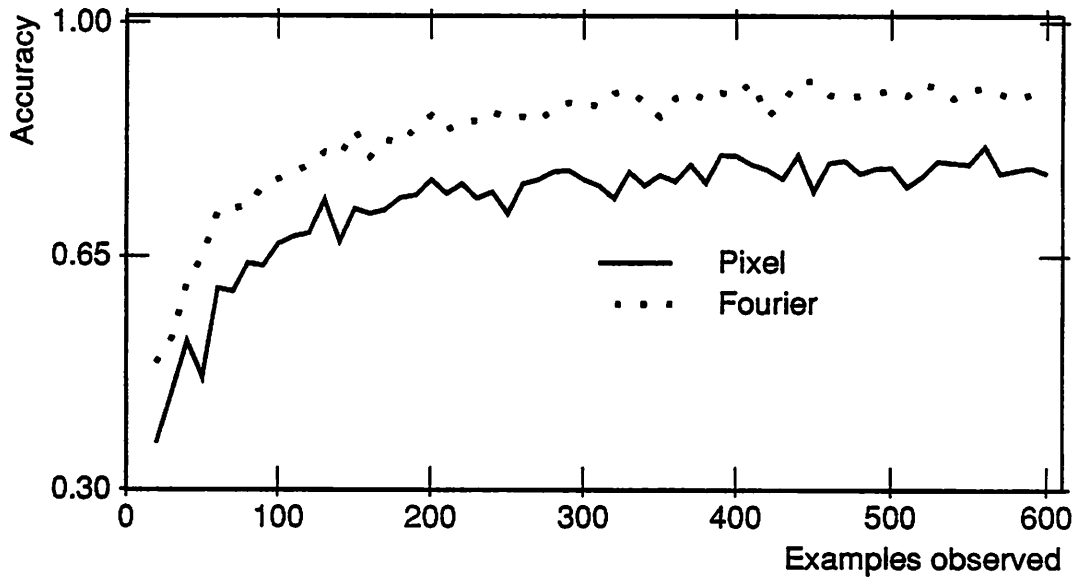


Figure 1.1 The difference in the ability of ID3 to generalize with two representations of handwritten characters.

in classifying a disjoint set of handwritten characters using the decision trees produced by ID3. Chapter 4 gives additional details about the conditions under which Figure 1.1 was obtained. This figure indicates that ID3 produces a more accurate decision tree with the Fourier representation than with the Pixel representation. Therefore, the Fourier representation is better than the Pixel representation if one wants to build a handwriting recognizer from the examples used in Figure 1.1.

The objective of this research is to understand the phenomenon illustrated in Figure 1.1. Specifically, we want to answer the following questions:

1. *Why* does instance representation affect generalization?

2. Given some alternative representations of a task, *how* can one identify the instance representation that will enable the learning algorithm to produce the most accurate hypothesis from the examples.

Answers to the above questions will increase our understanding of instance representations, which may result in techniques for designing good instance representations.

1.1 Methodology

The approach followed in this dissertation to answer the above questions is to test a hypothesis about why instance representation affects generalization. The hypothesis is that:

The instance representation that enables the examples to be specified with the fewest number of bits results in the most accurate model of the process that generates the examples.

This hypothesis is based on a number of previous results that show that data-compression and generalization are the dual of each other. These results are reviewed in Chapter 2. They show that in a variety of situations, given the examples in a particular representation, the model for the examples that enables the observed examples to be described most compactly makes the most accurate predictions about the unobserved examples. In this dissertation we show how this duality can be extended to situations where alternative representations for the examples are known.

The duality between data-compression and generalization suggests that one should be able to rank representations by measuring the minimum number of bits required to describe the examples in the different representations. Furthermore, if one can estimate the minimum number of bits required to describe the examples without observing a large number of samples of different sizes, then one can rank representations without having to estimate the performance of the learning algorithm on samples of different sizes. By estimating the performance of the learning algorithm on samples of different sizes one can rank representations by recording the highest estimated accuracy obtained with each representation. However, in order to determine the highest accuracy obtained with each representation, one must try samples of different sizes because in general there is no guarantee that the accuracy of the hypotheses produced by the learning algorithm increases by increasing the size of the sample from which the hypotheses are produced. Often, it may happen that increasing the sample size may reduce the accuracy of the resulting hypotheses. Learning algorithms that can over-fit the data by producing a hypothesis that correct on any set of examples are especially prone to exhibit this behavior. In addition, even if the accuracy of the hypotheses increases with sample size, it is possible to obtain incorrect ranking of the representations based on the estimated performance from a single sample. This is because the rate of increase may be different with different representations. Based on the performance estimated from a single sample one may conclude that one representation is better than the other, however on taking larger samples the accuracy of the hypotheses produced from samples in the representation considered to be worse may increase much more than the accuracy of the hypotheses produced from samples in the representation considered to be better.

A heuristic algorithm to compare instance representations was developed to test the above hypothesis. This algorithm, called ACR, attempts to solve the following problem:

Given: 1. A set of examples and alternative instance representations for the examples.

2. An algorithm for learning from examples.

Determine: The instance representation that enables the learning algorithm to produce the hypothesis that makes the most accurate predictions about the unobserved examples.

We call this problem the *representation comparison* problem. To solve this problem, ACR estimates the minimum number of bits required to express the examples in each instance representation. The representation that enables the examples to be described with the fewest bits is considered to be the best in the set of representations being compared.

To test the effectiveness of ACR, it was used to solve eighteen instances of the representation comparison problem. The predictions of ACR were tested by directly estimating the accuracy of the hypotheses produced by the learning algorithm from samples of different sizes drawn from the examples in the different representations, as in Figure 1.1. ACR's prediction was considered correct if the representation predicted to be the best by ACR did produce the most accurate hypothesis from the examples.

1.2 Overview of the Results

The conclusion of this dissertation is that the minimum number of bits required to describe the examples is an effective measure of the suitability of an instance representation for a learning algorithm. In particular, we have found ACR to be very effective in ranking representations. ACR was used to rank representations of ten tasks for ID3 and eight tasks for PT1, the preceptron tree learning algorithm (Utgoff, 1989). Together with the theoretical analysis of Chapter 2, the results obtained from these experiments support the conclusion that the compressibility of the examples can be used as a basis for ranking representations.

In addition, for the above eighteen problems, by using ACR we have been able to obtain a savings of 4 to 70 times over ranking representations by directly estimating the accuracy of the hypotheses that result from different representations. These savings were obtained over an inexpensive, though potentially inaccurate, method of accuracy estimation. Savings of 4 times were obtained when all the representations being compared were well-suited for the task but one representation was slightly better than the other. Savings of 70 times were obtained when one representation was clearly better than the other. If a more expensive, though potentially more accurate, method of accuracy estimation is used, then the savings obtained by using ACR to rank representations will increase. This suggests that not only is ACR effective in ranking representations, but also that it is faster to rank representations by using ACR than by directly estimating the accuracy of the hypotheses produced by the learning algorithm. In addition, ACR does not suffer from problems that arise when one tries to estimate the error-rates from a limited number of examples. This was observed in problems where the simple method of estimating the accuracy of a hypothesis based on a singe testing set gave unreliable results, and we had to resort to expensive resampling

methods to rank the representations by estimating the accuracies directly. ACR on the other hand correctly predicted the better representation for these problems based on the given set of instances.

The conclusion that one representation is better than another because it improves the compressibility of the examples suggests that different methods for improving the compressibility of the examples should result in different techniques for improving representations. Chapter 5 analyzes the reason for the improved compressibility of the examples with the better representation. This analysis shows how different methods for improving compressibility result in different types of representation change.

1.3 Contributions of this Research

There are two scientific contributions of this research. First, it is a further explication of the duality between data-compression and generalization. It shows how this duality can be utilized to rank representations for learning. Second, it provides an improved understanding of instance representations. It shows that one instance representation is better than another because the better representation enables the examples to be described more compactly. This in turn shows how different methods for improving compressibility can give rise to different techniques for improving a representation.

The engineering contribution of this research is ACR as a tool for comparing representations. An algorithm to compare representations has many applications. First, in situations involving automatic rule extraction from examples, only the hypothesis from the best instance representation need be used for classifying the future instances. The hypotheses produced from other representations can be rejected safely, alleviating the danger of making predictions with less accurate hypotheses. Second, while learning incrementally from a large set of labeled examples, one can begin learning in all the known representations and stop learning in representations that get identified as being worse than the others.

An algorithm to compare representations also helps in automating and facilitating the design of good instance representations. Currently, the design of a good instance representation is an art, and often representation design is the most time-consuming step in building a learning system. It took Quinlan (1983) two man months to design a good set of features for chess end-games; and with this set of features he could learn the concept of win in 3-ply in a few CPU seconds. An algorithm to compare representations can form the test component of an algorithm for searching the space of representations in a generate-and-test manner.

An algorithm to compare instance representations also provides an evaluator for *constructive induction* algorithms. The goal of constructive induction is to produce good instance representations by either automatically designing good initial vocabulary for the learning task (Callan & Utgoff, 1991) or improving an initial vocabulary

(Fawcett, 1991; Pagallo & Haussler, 1990; Matheus, 1990). Constructive induction algorithms require some method to determine whether their actions produce representations that result in accurate hypotheses. An algorithm to rank instance representations provides one method for determining whether a constructive induction algorithm is making progress in producing a good representation. Though ACR is a heuristic algorithm to compare representations, its effectiveness in ranking representations suggests that it provides a tool that can be used for the above applications.

1.4 A Guide to the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 develops the theoretical basis for the hypothesis that representations can be ranked by estimating the *minimum* number of bits required to express the examples with the different representations. This chapter reviews theoretical results from a number of areas and provides the missing links required to arrive at the above hypothesis.

Chapter 3 explains ACR in detail and analyses its complexity. It illustrates the operation of ACR with a small example.

Chapter 4 show the results obtained by using ACR to rank representations. This chapter also illustrates the difficulties faced in ranking representations by directly estimating the error-rates of the hypotheses produced with different representations, and the savings obtained by using ACR rather than methods that rank representations by directly estimating the error-rates.

Chapter 5 shows how different methods of improving compressibility can result in different techniques for producing a good representation. It identifies two distinct types of representation change by analyzing the reason for the improved compressibility of the examples with the better representation.

Finally, Chapter 6 concludes with a discussion of the contributions of this dissertation and directions for future research.

Appendices A, B, and C give detailed proofs of some of the results that form the theoretical basis for using compressibility as a measure of the suitability of a representation. Appendix D gives the values for the different parameters used in the experiments for evaluating ACR's performance.

CHAPTER 2

THE DUALITY BETWEEN DATA-COMPRESSION AND GENERALIZATION

This chapter explains the duality between data-compression and generalization. Section 2.1 discusses the duality informally. Sections 2.2, 2.3, and 2.4 review the theoretical results that establish this duality formally. Section 2.2 reviews the application of *algorithmic information theory* to inductive inference that establishes this duality when any program for a universal Turing machine is allowed to be a hypothesis for the examples. However, practical learning algorithms do not permit all possible programs to be hypotheses. Section 2.3 reviews results from *learnability theory* that establish the duality for a class of learning algorithms. Section 2.4 reviews the *minimum description length* principle that establishes this duality for statistical estimation problems. In these problems one is not learning functions or concepts, but instead one is modeling probabilistic phenomena. Section 2.5 reviews other efforts that have used data-compression to address issues that arise in learning from examples.

The body of this chapter discusses how each of the theoretical results establish the duality between data-compression and generalization. The implications of each of these results for the representation comparison problem are also discussed. The detailed proofs for the theorems can be found in the appendices and in the citations provided in the body of the chapter. The ideas presented in this chapter have been taken from a number of sources, which are cited in the text. However, the specific details presented here may differ from those in the original source.

2.1 Informal Justifications

As a simple model of the task of prediction, suppose that an observer is seeing a sequence of bits and is required to predict whether the next bit will be a zero or a one. The observer can predict the future bits successfully only if there is some redundancy in the observed sequence. If every segment of the string is totally unlike any other segment, there can be no basis for prediction. However if there is redundancy in the observed string, then the observer should be able to compress the string into a form that is more compact than just an enumeration of the bits. The regularities present in the string can be utilized to describe the string compactly. That is, predictability

implies redundancy which in turn implies compressibility. Denoting implication by $\Rightarrow$, we can summarize the above informal reasoning as:

$$\text{Predictability} \Rightarrow \text{Compressibility} \tag{2.1}$$

This is essentially the statement of the contrapositive of Martin-Löf's (1966) result that uncompressible strings pass all possible effective tests for randomness.

In addition to being sufficient for compressibility, redundancy is also necessary for compressibility. If a string is not redundant then every segment of the string is unlike any other segment, and the shortest description of the string is just an enumeration of the bits. Therefore, if a string is compressible then there must be some pattern in it that is used to describe the string compactly, and this pattern can form the basis for future predictions.

Intuitively, the reason why a compact hypothesis for the examples has a high predictive accuracy is that only a few hypotheses are expressible in a small number of bits. As a result, there is a high likelihood that a randomly chosen set of examples will not be consistent with any of these hypotheses. Consequently, if a compact hypothesis *is consistent* with a sufficiently large number of randomly chosen examples, then one can have a high confidence that the hypothesis is a good approximation of the concept generating the examples. It is very unlikely that by chance alone one would find a compact hypothesis consistent with the examples. This reasoning is similar to the one used in significance tests in statistical hypothesis testing where one rejects a hypothesis if one observes an event that has a small probability if the hypothesis were to be true. In summary, compressibility implies redundancy which in turn implies predictability. This observation can be stated symbolically as:

$$\text{Compressibility} \Rightarrow \text{Predictability}. \tag{2.2}$$

Precise formulation of the above informal discussion forms the basis of many formal arguments for showing that in a number of situations finding a compact hypothesis suffices for generalizing from examples (Kemeny, 1953; Pearl, 1978; Blumer, Ehrenfeucht, Haussler & Warmuth, 1987). Combining (2.1) and (2.2) gives:

$$\text{Predictability} \Leftrightarrow \text{Compressibility}$$

2.2 Algorithmic Information Theory

Algorithmic information theory is the study of the size of the shortest program required to compute a function on a universal Turing machine. This theory was originated by Solomonoff (1964) to formalize inductive inference and by Kolmogorov (1965,1968) and Chaitin (1977) to formalize the notion of a random string.

Suppose that one is faced with the problem of extrapolating a string, and a number of hypotheses for the function generating the string are known. That is, we are in a

situation of predicting a string generated by a deterministic rule, or a function. In order to make the most accurate predictions, one would like to select the hypothesis that is most likely to be correct.

Bayes rule, also known as the probability inversion formula, is a method of updating the prior probability of the hypothesis H being correct in light of the observations D. Bayes rule says that:

$$P(H \mid D) = \frac{P(D \mid H)P(H)}{P(D)}.$$

In this formula, $P(D \mid H)$ denotes the conditional probability of D given H. It is the probability that the observations D would have occurred if H were the rule generating the observations. $P(H)$ is the prior probability of H being correct. $P(D)$ is a normalizing factor that gives the probability that the observations D would have occurred, no matter which hypothesis was correct. If one has to choose between two hypotheses H_1 and H_2 on the basis of the same observations D then $P(D)$ is a constant and,

$$\frac{P(H_1 \mid D)}{P(H_2 \mid D)} = \frac{P(D \mid H_1)P(H_1)}{P(D \mid H_2)P(H_2)}.$$

Now if the observations D are inconsistent with a hypothesis H, then the probability of observing D when H is the correct hypothesis is zero. That is, for inconsistent hypotheses $P(D \mid H) = 0$. Therefore, one can immediately discard inconsistent hypotheses from consideration as they cannot be the rule generating the observations. Similarly, for any hypothesis consistent with the observations $P(D \mid H) = 1$, because if this hypothesis were true, one would have definitely observed D. Consequently,

$$\frac{P(H_1 \mid D)}{P(H_2 \mid D)} = \frac{P(H_1)}{P(H_2)}.$$

That is, the consistent hypothesis with the greatest prior probability has the greatest probability of making a correct prediction.

The above discussion shows that we can select the most accurate hypothesis if we know the prior probability of various hypotheses. This raises the problem of determining the prior probabilities of hypotheses. To solve this problem, Solomonoff's (1964) idea was to assign prior probabilities based on the length of programs required to compute a hypothesis on a universal Turing machine (UTM). The reason for choosing a UTM as the basis for determining the prior probabilities is that by the Church-Turing thesis any function that is intuitively considered to be computable can be computed on a UTM, and for making predictions only computable hypotheses need be considered. The rest of this section discusses how the length of the shortest program to compute a hypothesis on a UTM can be used to obtain an accurate estimate of the prior probability of a hypothesis.

Consider a 3-tape universal Turing machine U with a one-way read-only program tape, a one-way output tape, and a two-way read-write work-tape. Such a 3-tape universal Turing machine will be called a *prefix machine*. These machines have the property that no program for which the machine halts can be a prefix of any other program. Prefix machines are used to define prior probabilities rather than a standard universal Turing machine because prior probability of a hypothesis is based on the sum of the lengths of different programs for computing the hypothesis on a UTM, and for a standard UTM this sum may not be finite. The use of prefix machine will guarantee that this sum is less than one for any hypothesis. This point is discussed further below and in Appendix A. If U halts with a program p, then let $U(p)$ denote the contents of the output tape. Denote by $l(p)$ the number of symbols of the alphabet of the machine required to write p, that is, $l(p)$ is the length of the program p. Without loss of generality consider only prefix-machines over the binary alphabet $\{0,1\}$. Programs in any other alphabet, say of size r, can be encoded in the binary alphabet with an increase in the size of each program by a multiplicative factor of $\log_2(r)$. For the purposes of comparing two hypotheses, this uniform increase does not matter. Henceforth all logarithms are binary, and the subscript 2 will be dropped from the logarithms.

Assign to a string s the prior probability

$$P_U(s) = \sum_{p \in \{0,1\}^* : U(p)=s} 2^{-l(p)}.$$

The summation in the above equation is over all programs p for which the prefix-machine U halts and writes s on the output tape. It is justified to call $P_U(s)$ a probability because U is a prefix machine and therefore the above sum is defined and is less than 1. Li & Vitányi (1989) call $P_U(s)$ the *Solomonoff-Levin distribution*, and Chaitin (1977) calls it the *algorithmic probability* of s. The algorithmic probability $P_U(s)$ can be interpreted as follows. Consider each program for the prefix machine U to be produced by flips of an unbiased coin. This process produces a program p of length $l(p)$ with probability $2^{-l(p)}$. Therefore, the probability of producing a program for the string s is the sum of the probabilities of producing the different programs that generate s. Algorithmic probability can be considered to be a combination of two well known tenets for induction. The first is the *Occam's Razor* principle which says that between two hypotheses that agree equally well with the data choose the simpler hypothesis. If simplicity is measured by the size of programs required to compute a string, then short programs contribute a larger amount to algorithmic probability than longer programs. The second is the principle of *insufficient reason*, also known as the principle of *indifference*. This says that if the available evidence gives no reason to consider either of H_1 or H_2 more likely, then they should be assigned equal probabilities (Solomonoff, 1964; Jaynes, 1988). This rule is incorporated in $P_U(s)$ by making programs of equal length contribute an equal amount to the algorithmic probability of a string.

The Turing machine U used to define algorithmic probability is a universal machine. Therefore, it is possible that for some programs p, U does not halt. As a result, $\sum_s P_U(s) \leq 1$. However, algorithmic probability satisfies all other laws of probability. Real-valued functions that satisfy all laws of probability except that the sum of the value of the function over its domain is less than or equal to one will be called *measures*. A simple result from information theory, called *Kraft's inequality*, shows that algorithmic probability is a measure. Kraft's inequality is proved in most introductory texts on information theory (Abramson, 1963; Blahut, 1987). Appendix A gives a proof for Kraft's inequality and the fact that algorithmic probability is a measure.

Arguments similar to those of Levin (Zvonkin & Levin, 1970) and Chaitin (1975) are used in Appendix B to show that if $K_U(s)$ is the size of the shortest program required to compute the string s on a prefix-machine U, then there exists the additive constant c, which depends only on U and not on s, such that for an $\epsilon \leq c$:

$$\log P_U(s) = K_U(s) \pm \epsilon$$

The shortest program to compute a function will be called the *minimal program* for that function. The above equation implies that for any two functions for which the size of the minimal program for the prefix-machine U differs by at least $2c$, the function with the shorter minimal program will have a greater algorithmic probability. However, if the difference is less than $2c$ then the size of the minimal program does not tell which hypothesis will have a greater algorithmic probability. $K_U(f)$ is called the *Kolmogorov complexity* or the *algorithmic information* of s.

Notice that we chose to call $K_U(s)$ the algorithmic information, rather than the algorithmic information with respect to U. This is because the size of the minimal program is relatively insensitive to the exact choice of the prefix machine. More precisely, given two prefix machines U_1 and U_2, there exists an additive constant c that depends only on U_1 and U_2, such that for an $\epsilon \leq c$,

$$K_{U_1}(s) = K_{U_2}(s) \pm \epsilon.$$

This is because U_1 and U_2 are universal Turing machines and they can simulate each other. By appending a fixed program to all programs of U_2, U_1 can simulate U_2 and vice-versa. To see this, say c_1 is the size of the translator (interpreter) from U_1 to U_2, and let c_2 be the size of the translator from U_2 to U_1. If x is the size of the minimal program for s on U_1, then the size of the minimal program for s on U_2 cannot be less than $x - c_2$, because otherwise one can get a program of size less than x for U_1, which contradicts the fact that x is the size of the minimal program. Also, the size of the minimal program for U_2 cannot be greater than $x + c_1$, because translating x for U_2 gives a program of size $x + c_1$. Therefore,

$$K_{U_1}(s) = K_{U_2}(s) \pm \max(c_1, c_2).$$

This result is known as the *invariance theorem* (Solomonoff, 1964; Kolmogorov, 1965). This theorem justifies dropping the subscript U in the definition of algorithmic information and algorithmic probability. It implies that for hypotheses for which the

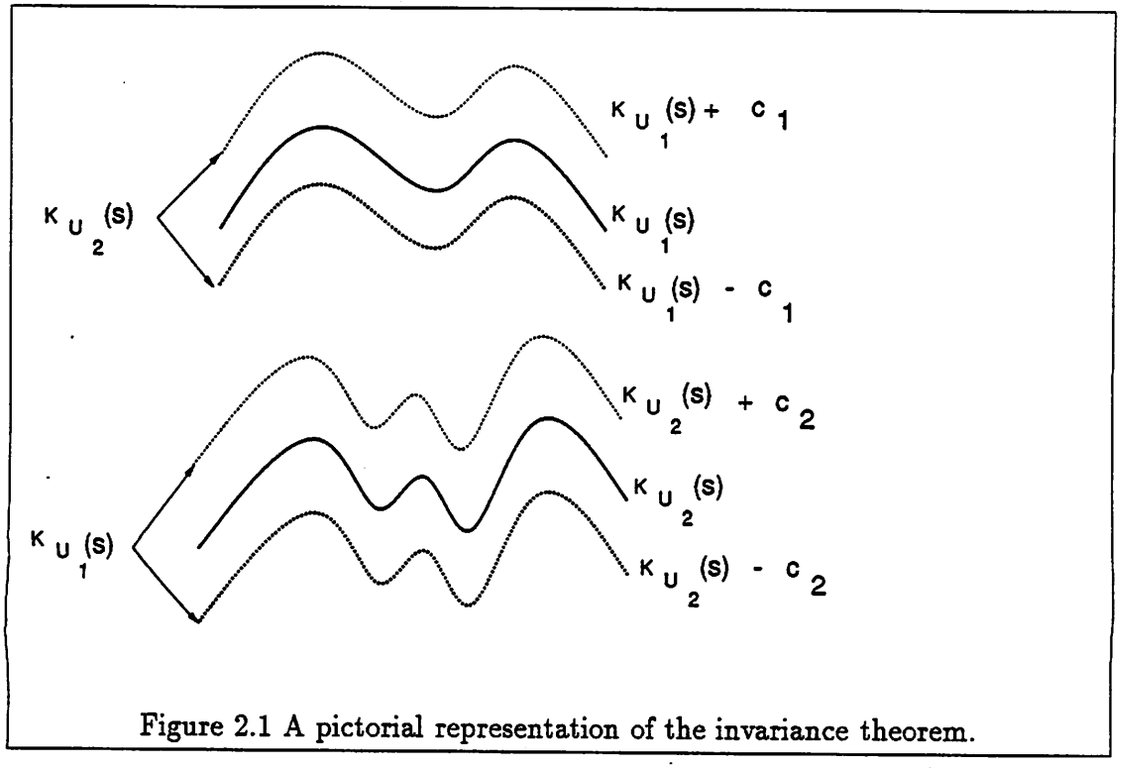


Figure 2.1 A pictorial representation of the invariance theorem.

size of the minimal program is greater than a fixed amount, it does not matter which prefix-machine is used to assign prior probabilities. Figure 2.1 is a pictorial representation of the invariance theorem.

In the above discussion and in the discussion that follows, we often choose to ignore additive constants required in the various equations. For instance, algorithmic information and the negative logarithm of the algorithmic probability are equal only up to an additive constant that depends on the particular prefix machine employed for defining the above quantities. Similarly, the algorithmic information defined by using two different prefix machines is equal only up to an additive constant that depends on the two machines. Strictly speaking, ignoring these constants is justified only when conclusions are drawn about the asymptotic behavior of algorithmic information. In the rest of this section we shall restrict attention to the asymptotic case and not mention it explicitly.

The above discussion shows that it is sufficient to choose the hypothesis with lower algorithmic information *if* algorithmic probability is the prior distribution that governs the accuracy of the various hypotheses. But, why should algorithmic probability, rather than some other distribution govern the prior probabilities?

Levin (Zvonkin & Levin, 1970) proved a theorem that states the following: except for hypotheses with very small prior probabilities, algorithmic information produces a good approximation of the probability assigned to a hypothesis by *any* computable probability measure. This implies that if one chooses a hypothesis on the basis of

algorithmic information, and the chosen hypothesis does not have a very poor predictive accuracy, then with high likelihood the choice will coincide with the hypothesis that would have been chosen based on the actual prior distribution. However, since the actual prior distribution is not known, algorithmic information forms a good basis for choosing hypotheses. In this sense, algorithmic information is a *universal prior distribution*, it can be used to approximate any prior distribution. Appendix C gives a proof of this theorem.

One might argue that since algorithmic information is a good approximation of only computable prior distributions, it is not applicable in situations where an uncomputable prior distribution governs the probability of a hypothesis being correct. However, notice that in such situations there cannot be any algorithmic method *at all* to choose between consistent hypotheses. If there were such a method, then by choosing the examples to be consistent with only two hypotheses of interest, one could determine which hypothesis has a higher prior probability based on which hypothesis is chosen by the proposed method. This then gives a computable method of determining the prior distribution, contrary to the claim that an uncomputable prior distribution is determining the probability of a hypothesis being correct. Therefore, for a particular problem, if there is any algorithm *whatsoever* to choose among consistent hypotheses, then the hypothesis chosen on the basis of algorithmic information will be the same, except for hypotheses that have poor predictive accuracy.

The fact that algorithmic information gives rise to a universal prior distribution provides the strongest justification of data compression view of generalization. It shows that, asymptotically, the best predictions are obtained by using the hypothesis that can be computed with the shortest program.

To summarize, this section started with the observation that the Bayes rule implies that the consistent hypothesis with the greatest prior probability will make the most accurate predictions. Then, algorithmic probability was defined, which assigns prior probabilities based on the size of programs required to compute a hypothesis on a universal Turing machine. Then it was shown that for any hypothesis the logarithm of the algorithmic probability and algorithmic information are equal up to an additive constant. This helps to show that algorithmic probability is relatively insensitive to the choice of the particular universal Turing machine. Finally, it was shown that in most cases, asymptotically in the amount of algorithmic information, the hypothesis chosen by using algorithmic information will coincide with the hypothesis chosen by any computable method that can identify the hypothesis with a greater predictive accuracy.

To see the implication of algorithmic information theory for the representation comparison problem, consider what happens when one changes the representation. Changing the representation changes the function that produces the examples. Suppose that with representation R_1 one needs to learn the function f_1 and with the representation R_2 one needs to learn the function f_2. Say l_1 is the size of the shortest program for the set of examples drawn from f_1, and l_2 is the size of the shortest

program for the set of examples drawn from f_2. Suppose these programs correspond to hypotheses H_1 and H_2 respectively. Assuming that none of these hypotheses has a very small prior probability, then asymptotically in l_1 and l_2, H_1 is the most accurate hypothesis when the examples are represented in R_1 and H_2 is the most accurate hypothesis when the examples are represented in R_2. The prior probability of H_1 being correct is 2^{-l_1} and that of H_2 being correct is 2^{-l_2}. If $l_1 < l_2$, that is, the size of the shortest program when the examples are represented in R_1 is less than the size of the shortest program when the examples are represented in R_2, then H_1 is more accurate than H_2. That is, the representation R_1 results in a more accurate hypothesis than the representation R_2.

There are two problems with using algorithmic information for representation selection. First, the above results are valid only asymptotically. Second, algorithmic information is uncomputable in general. Therefore, in general, the hypothesis with the least algorithmic information cannot be identified. One approach to address this problem is to restrict the class of programs considered as hypotheses. The next section shows that under certain circumstances, even when the hypotheses are described in a restricted concept description language, the most compact hypothesis is the one that enables the examples to be described most compactly.

2.3 Learnability Theory

The previous section showed how representations could be ranked if one could determine the hypothesis with the least algorithmic information. Unfortunately, because of the asymptotic nature of the results and the uncomputability of algorithmic information, this approach cannot be used in practice. However, learnability theory establishes the duality between data-compression and generalization for an idealized model of learning algorithms that generalize by finding a compact hypothesis that is correct on all the observed examples, such a hypothesis is called a *consistent* hypothesis.

Consider a learning algorithm that searches the space of hypotheses for a consistent hypothesis in strict order of increasing complexity. That is, given a set of examples it first tries to find a consistent hypothesis that can be expressed in c_0 bits, if it cannot find such a hypothesis it looks for a consistent hypothesis that can be expressed in c_1 bits ($c_0 < c_1$), and so on. Call such a learning algorithm an *ideal* learning algorithm. Searching the space of hypotheses in a strict order of increasing complexity is a good model of a number of learning algorithms that generalize by producing a compact hypothesis consistent with the observations. Examples of such algorithms are ID3 without pruning (Quinlan, 1986) and FRINGE (Pagallo & Haussler, 1990) which generalize by building compact decision trees for the observations, and AQ11 (Michalski & Chilausky, 1980) which finds a small disjunctive normal form expression for a given set of observations. These algorithms start with a simple hy-

pothesis and make the hypothesis increasingly more complex in order to be consistent with the observations.

Suppose that the ideal algorithm has observed m examples chosen independently according to a fixed but arbitrary probability distribution on the examples, and for these m examples it finds a consistent hypothesis h that can be expressed in s bits. Following the development of Blumer et al. (1987), consider the chances that h misclassifies a randomly chosen example with probability greater than ϵ. That is, we would like to determine the probability that a hypothesis with *error-rate* greater than ϵ is correct on m independently chosen examples. Call such a hypothesis *insidious*. Since the probability that an insidious hypothesis is correct on one example is less than $1 - \epsilon$, and the m examples are independent, the probability that h is insidious is bounded by:

$$P_{\text{insidious}}(h) \leq (1 - \epsilon)^m.$$

Now consider the probability that any hypothesis chosen from a class H of hypotheses is insidious. This probability is bounded by:

$$P_{\text{insidious}}(H) \leq \mid H \mid (1 - \epsilon)^m \tag{2.3}$$

where, $\mid H \mid$ is the number of hypotheses in H.

To see why the size of the hypothesis space is needed in (2.3), suppose that for some reason getting an even number in a roll of a die is an undesirable event. The probability of getting a particular even number is 1/6. The probability of an undesirable event occurring in a roll of a dice is:

$$P_{\text{undesirable}} = P(2) + P(4) + P(6)$$

In general,

$P_{\text{undesirable}} \leq$ number of undesirable events $\times$ P(the most likely undesirable event)

which is bounded by:

$P_{\text{undesirable}} \leq$ total number of events $\times$ P(the most likely undesirable event).

Denote by $L(s)$ the number of hypotheses in the concept description language employed by the ideal algorithm that can be expressed with at most s bits. Since the ideal learning algorithm searches the space of hypothesis in the order of increasing complexity, if it finds a consistent hypothesis that can be expressed with s bits, then it has found a hypothesis from a class of at most $L(s)$ hypotheses. Therefore, the probability that any hypothesis chosen from this class is insidious is:

$$P_{\text{insidious}}(H) \leq L(s)(1 - \epsilon)^m$$

For a number of concept description languages, more hypotheses can be described if one is given more expressive power. That is, $L(s)$ is non-decreasing in s. Therefore, the above bound on obtaining an insidious hypothesis establishes two facts, first

for a given set of examples as the hypotheses get more compact (s decreases), the probability of obtaining an insidious hypothesis decreases. Second, for a given set of examples and a given probability of obtaining an insidious hypothesis, compact hypotheses have a low error-rate. That is, producing a compact hypothesis suffices for obtaining with high confidence a hypothesis with low probability of error. Decision trees, feed-forward neural networks, boolean formulae and boolean circuits are examples of concept description languages in which increasing the expressive power increases the number of hypotheses that can be described. Pearl (1978) also discusses this relationship between the complexity and credibility of hypotheses.

Blumer et al. (1989) use the notion of the *VC-dimension* of a class of concepts to determine the minimum number of examples necessary to learn any member of the class. The VC-dimension of a class of concepts C is defined as follows. Given a set of n unlabeled examples, there are 2^n possible labelings of these examples as positive and negative examples of a concept. Some of these labelings may be consistent with members of C and some labelings may not be consistent with any member C. The VC-dimension of C is the largest number of examples such that for every possible labeling of these examples as positive and negative examples, there is a concept in C consistent with the labeling (Vapnik & Chervonenkis, 1971). Blumer et al. show that if one wants to ensure with high probability that a learning algorithm produces a hypothesis with low error-rate for every member of a class of concepts, then the number of examples required by the learning algorithm increases as the VC-dimension of the class of concepts increases.

Now suppose that the ideal algorithm cannot find a compact hypothesis for the examples, and that the hypothesis it did find requires k bits to be specified. That is, the examples are from one of $L(k)$ concepts. If the concept description language is such that increasing the number of concepts also increases the VC dimension of the space, then as k increases the VC-dimension of the space of concepts considered by the ideal learning algorithm also increases. This in turn implies an increase in the number of examples necessary to produce an accurate hypothesis with high confidence for every member of the class. Equivalently, for a fixed number of examples, the inability to produce a compact hypothesis implies that for at least one concept in the class, either one has a low confidence in the accuracy of the hypothesis that was produced, or the hypothesis has a low accuracy. The assumption that the VC-dimension of a class of concepts increases as the cardinality of the class increases is true for concept description languages that are rich enough to represent any function of the inputs. For instance, increasing the number of bits allowed to describe a decision tree, boolean formula, boolean circuit, increases the number of boolean functions that can be described and since each boolean function represents a new dichotomy of the examples as positive and negative instances, the VC-dimension of the class of concepts also increases.

Baum and Haussler (1989) show how the above discussion can be applied to loading neural networks, where loading a neural network means finding a set of weights

for the units such that network correctly classifies all the observed examples. For fixed precision weights, the number of bits required to describe a given neural network is of the order of the number of weights. They show that if a network can be loaded with a set of examples, then the fewer the number of weights in the network, the more confidence one has that the loaded network has the desired predictive accuracy. Furthermore, for a fixed number of examples and a desired confidence, the maximum accuracy obtainable from a network in the worst case decreases with the number of weights in the network. Therefore, if a set of examples cannot be loaded into a small network, and one does succeed in loading the examples into a larger network, then one is likely to have a poorer predictive accuracy than if one had succeeded in loading the examples into the small network.

As was discussed in the previous section, different instance representations produce different functions that need to be learned. If we restrict attention to the types of learning algorithms discussed above, then compactness of the hypothesis produced by the learning algorithm is a measure of the size of the space of possible concepts considered by the algorithm. Therefore, the instance representation that enables the learning algorithm to produce the most compact hypothesis makes the learning algorithm search the space with fewest concepts, and therefore it is most likely to result in an accurate hypothesis.

2.4 Minimum Description Length Principle

The *minimum description length* principle establishes the duality between data compression and generalization for statistical estimation problems (Rissanen, 1989; Wallace & Freeman, 1989). In these problems, the goal is not to learn a deterministic function, but rather to model a probabilistic phenomenon. A probabilistic phenomenon is one where identical values of the experimental parameters do not always produce identical outcomes. The outcome of an experiment depends upon the input parameters in a probablistic manner.

Intuitively, the minimum description length principle is derived from the observation that in modeling a probabilistic phenomenon one is trying to derive constraints on the occurrences of various outcomes. Now suppose that one's objective is to be able to describe any set of observations as compactly as possible. Any constraints on the outcomes imply that all outcomes are not equally likely, and therefore one can reserve short encodings for more likely outcomes. Different models in the class of models being considered will assign different probabilities to various outcomes. Since for each model one assigns short encodings to the more likely outcomes, the length required to specify any particular outcome will differ for different models. Among all possible models in the class, the one that allows the observations to be described most compactly considers the observations to be more likely than any other model in the class. Therefore, the model that enables the string to be described most compactly

is the maximum-likelihood model for the observations.

Rissanen (1989) has derived formulas for situations where one wants to select the best model from a parametric class of probabilistic models. In addition to estimating the value of the parameters, the number of parameters is also to be estimated from the observations. He shows that under certain smoothness conditions, the model selected by his minimum description principle is a generalized maximum-likelihood model. This is the model that assigns maximum probability to the observations among all possible models that differ not only in the values of the parameters but in the number of parameters as well.

2.5 Other Data Compression Based Approaches

In addition to the above formal explications, different aspects of the duality between data-compression and generalization have been pointed out by a number of authors. Kemeny (1953) discusses this relationship in the context of theory formation in science. Maciejowski (1979) proposes the use of simplicity as a model selection criterion when one does not have enough prior information to formulate a small class of models within which to search for the best model, but instead one has to utilize fairly general model classes; and in addition one has limited amount of observations available. Watanabe (1985) and Rendell (1986) point out that concept formation via induction is equivalent to information compression because one is trying to partition a large number of examples into a small number of classes.

Within machine learning there have been a few efforts to apply the data compression view to specific problems. Thorton (1988) shows how concept learning with tree structured attributes can be viewed as data compression. He defines *generalized data compression* to be the problem of finding a representation for the observations that does not exceed a fixed description length while preserving the maximum amount of information about the observations. He describes an algorithm for achieving this by clustering the data and representing the observations by the centroid of each cluster. By defining the distance measure between nodes of a tree structured attribute to be such that the distance between children of a node is always less than the distance between the children and non-children of the node, learning a concept from tree structured attributes can be viewed as generalized data compression.

Segan (1985) applies a *minimum representation length* criterion for learning conjunctive concepts when examples may be noisy. From the observations he estimates the probability that an observation belongs to a concept when a particular descriptor is true. From all possible subsets of descriptors, the subset with which the representation of the observations is the shortest is selected. The size of the representation is computed from the number of elementary operations in each descriptor, each elementary operation being assigned a fixed representation length. After finding such a subset for each concept, an instance is classified as belonging to the class for which

it has the highest probability.

Georgeff & Wallace (1985) propose that competing theories should be selected on the basis of a two part codelength. One part of the codelength is needed to specify a theory within the class of theories being considered, and the second part of the codelength is needed to describe the data using the chosen theory. They further suggest that domain knowledge may be used to define a *theory description* language and a *data description language* in a manner that the constructs in these languages that are more likely to occur have shorter descriptions. In particular, along with each construct of the language, they suggest recording the probability with which one expects the construct to occur. The negative logarithm of the probability gives the code length required to represent the construct. Total codelength required to express a set of observations is obtained by adding the codelength required to express each of the constructs. The theory that enables the observations to be described by the shortest codelength is considered to be the best theory. They further suggest that the above approach can be used in a hierarchical manner in which the theories formed at the lower level form the observations for the theories at the higher level.

Wolff (1982) uses data compression for learning context-free grammars from unsegmented, semantics free, text. His approach is to apply a set of data compression methods to the text in stages. For example, first repeated substrings within the text are found and proposed as words. Each occurrence of a word in the text is replaced by a pointer to the word. At the end of such a scan, different substrings that occur in the same context are replaced by disjunctive production rules. This compression method and additional compression methods for replacing substrings with non-terminals are used to form a grammar at one level. By replacing the terminal symbols at one level with non-terminals derived from the above process, a new string is obtained that becomes a candidate for compression. A grammar for the text is derived by recursively applying the compression methods to the new string.

Muggleton's (1987) Duce system for discovering concepts in propositional logic also works by attempting to compress a set of rules. His technique is to observe repeated patterns in the rules and to replace them with new rules. Different operators look for different types of patterns. At each stage, the operator that results in the largest reduction in the number of symbols is applied. A user assists the program by selecting or rejecting any new rules proposed by Duce.

2.6 Summary of the Chapter

This chapter presented a number of arguments that establish the duality between data-compression and generalization. Results from algorithmic information theory established this duality when any program for a universal Turing machine could be a hypothesis. Learnability theory established this duality for learning algorithms that search a space of hypotheses strictly in the order of increasing complexity. The

minimum description length results from statistical estimation theory established the duality for modeling probabilistic phenomena. In each case we also discussed the implication of these results for the representation comparison problem. The conclusion from this discussion is that in the above situations the representation that enables the examples to be encoded the most compactly will result in the most accurate hypothesis.

CHAPTER 3

AN ALGORITHM TO COMPARE INSTANCE REPRESENTATIONS

The informal discussion and the formal results of the previous chapter show that in a variety of situations the likelihood of a hypothesis being accurate increases with the compactness of the hypothesis. This suggests that in order to determine the representation that enables the learning algorithm to produce the most accurate hypothesis from the given examples, one can determine the minimum number of bits required to describe the classification of the examples when different representations are employed. The representation with which the classification of the examples can be described most compactly should, with high probability, result in the most accurate hypothesis.

We take the formal results quoted in the previous chapter as being suggestive of a criterion by which instance representations can be compared. However, these results are valid only with specific assumptions. In practice, it may be difficult to ascertain whether the assumptions hold exactly or only approximately. For instance, practical learning algorithms do not consider all possible programs as hypotheses. They usually have a fixed concept-description language. Therefore, strictly speaking, results from algorithmic information theory do not justify our use of compactness of a hypothesis as a basis for comparing representations. Also, the asymptotic nature of the results from algorithmic information theory make them unsuitable for application to actual problems. Similarly, even though a number of algorithms generalize by producing a compact hypothesis for the examples, one cannot say that these algorithms search the space of hypotheses in a strict order of increasing complexity. Therefore the results from learnability theory cannot be applied directly. Finally, we often compare representations for learning functions rather than for modeling probabilistic phenomena. In these situations, the various quantities required in the precise formulation of the minimum length description principle may not exist.

The next section presents ACR, an algorithm for comparing representations based on the duality between data-compression and generalization. We consider ACR to be a further explication of this duality, rather than being a direct application of any of the formal results mentioned above. The results obtained by using ACR suggest that it is effective in ranking representations in a variety of situations. The reason for ACR's effectiveness is its use of compressibility as a measure of the suitability of

an instance representation for a learning algorithm. Estimating the minimum number of bits required to describe the examples enables ACR to rank representations without directly estimating the accuracy of the hypotheses produced by the learning algorithm. Estimating these accuracies can be expensive, especially when a limited number of examples are available. In these situations, one needs to ensure that a representative set of test examples has been produced from which the accuracy of a hypothesis can be estimated reliably. Expensive resampling methods are usually required to produce such test sets (Weiss & Kulikowski, 1991). Also, by using the number of bits required to describe a hypothesis as the basis for ranking representations, often ACR can rank representations without observing samples of all possible sizes. Tests that observe the trend in the number of bits required to describe the examples, and estimate whether the number of bits can be reduced by increasing the sample size, enable ACR to rank representations based on small samples from the examples. Section 3.1 gives the details of the design and the current implementation of ACR. Section 3.2 gives a detailed example of the steps involved in using ACR to rank representations.

The current implementation of ACR only handles problems where the task is to learn to classify a set of objects as belonging to one of finite number of classes. Also, the current implementation ranks two representations at a time. More than two representations are ranked by pair-wise comparison of the representation considered to be best so far with one other representation. Section 3.3 discusses extensions to the current implementation of ACR that enable it to overcome these limitations. Section 3.4 analyses the worst case complexity of the current implementation of ACR.

3.1 ACR: The Design and Implementation

ACR works by estimating the minimum number of bits required to describe a consistent hypothesis for the examples with each representation. The number of bits required to specify a consistent hypothesis will be called the *codelength* of the hypothesis. ACR considers the instance representation for which its estimate of the minimum codelength is the smallest to be the best in the set being compared.

To estimate the minimum codelength for the examples, ACR observes the trend in the codelength as an increasing number of examples are presented to the learning algorithm. Given a sample from the examples, the learning algorithm produces a hypothesis that is correct on some examples and incorrect on others. To make this hypothesis consistent, ACR imposes an arbitrary order on the examples and records a 0 for all examples for which the hypothesis is correct and a 1 for the other examples, along with the appropriate correction. The list marking the position of errors is called the *error-list*. For instance, suppose the examples are:

$$< -(x1 = 0), (x2 = 0), (x3 = 0) >,$$

$$< +(x1 = 0), (x2 = 1), (x3 = 0) >,$$
$$< +(x1 = 1), (x2 = 0), (x3 = 0) >,$$
$$< -(x1 = 1), (x2 = 1), (x3 = 0) >,$$
$$< +(x1 = 0), (x2 = 1), (x3 = 1) > .$$

Suppose that based on a random sample from these instances the learning algorithm produces a hypothesis that says, "if $x1 = 1$ then classification is a $+$, otherwise the classification is a $-$", then the error-list is [0, 1, 0, 1, 1]. If by drawing a different random sample the hypothesis changes to $+$ if $x2 = 1$ and $-$ otherwise, the error-list becomes [0, 0, 1, 1, 0]. Notice that if the samples are drawn at random, and if the hypothesis produced by the learning algorithm changes with the sample, the presence of an error at a particular position is a random event, or a random-variable, whose value depends on the hypothesis.

For a given ordering of the examples, the hypothesis produced by the learning algorithm augmented with the location and the value of the corrections is a consistent hypothesis. The consistent hypothesis says, "take the examples in the order that they appear in the list of examples, classify them with the hypothesis produced by the learning algorithm, and correct the classification of the positions where the error list has a 1 to be the specified values". The codelength required to specify this hypothesis is the sum of the number of bits required to specify the original hypothesis and the number of bits required to specify the corrections. The length of the description of the sentence "take the examples in the order that they appear in the list of examples" is a fixed overhead, which can be ignored in the comparisons. This procedure is the one used by Quinlan & Rivest (1989) for constructing decision trees using the minimum description length principle and it is similar to Rissanen's (1989) and Wallace & Freeman's (1989) two part encoding.

As the sample size increases, the learning algorithm may produce hypotheses that are correct on a larger number of examples. This decreases the number of bits required to describe the errors. However, it may increase the number of bits required to specify the hypothesis. One can estimate the minimum codelength required to specify a consistent hypothesis without observing samples of all possible sizes in the following situations:

1. It is established that increasing the sample size increases the codelength. This is a symptom of the situation that the learning algorithm cannot utilize the regularities present in the examples any further in order to produce a compact hypothesis for the examples.

2. It is established that there are no regularities left in the examples, and so presenting additional examples to the learning algorithm will not reduce the codelength further.

Section 3.1.2 and Section 3.1.3 describe tests that provide an indication of these

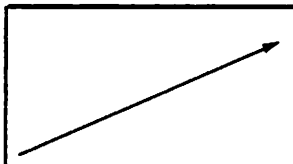
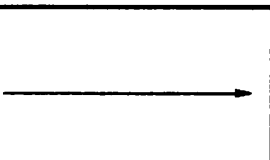
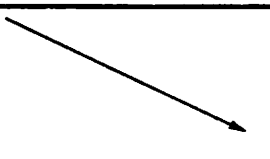
Codelength vs. Sample-size			
Indicates	The learning algorithm is unable to extract the remaining regularities in the examples.	No unextracted regularities remain in the examples.	The learning algorithm is extracting the regularities in the examples.
Conclusion	Presenting more examples will not reduce the codelength.	Presenting more examples will not reduce the codelength.	Presenting more examples can reduce the codelength
Action Indicated by	Terminate the run.	Terminate the run.	Continue the run.
	Rank test.	Independence test.	Failure on both the rank test and the independence test.

Figure 3.1 Possibilities considered by ACR after each sample

conditions. Figure 3.1 illustrates the options considered by ACR after each sample. Given these tests, the algorithm to compare two representations $R1$ and $R2$ is:

1. In a *run*, draw random samples of increasing size from examples in each representation and present them to the learning algorithm. For each representation, terminate a run if either of the two tests mentioned above indicates that the codelength is unlikely decrease by drawing larger samples of examples in that representation.

2. A single run may be based on unrepresentative samples, therefore conduct multiple runs in order to rank the representations with a desired confidence.

In effect, ACR estimates the minimum codelength by observing the behavior of the learning algorithm on a small number of small samples. In order to estimate the minimum codelength, ACR conducts tests to determine whether the codelength is likely to decrease by taking larger samples. It terminates the run if either of the two tests indicate that the codelength is unlikely to be reduced by taking more examples.

Specifically, a run returns the pair $(v1, v2)$, which is $(1, -1)$ if $R1$ is better, $(-1, 1)$ if $R2$ is better and $(0, 0)$ otherwise. Ignoring the tied values of $v1$ and $v2$, if in repeated runs the decision about the better representation changes then $v1 - v2$ is equally likely to be 2 or -2. A large number of positive or negative values indicates that one representation is better than the other. After each run, a *sign test* determines whether there is a significantly greater proportion of positive values or negative values

seen so far (Gibbons, 1971). The hypothesis that both representations are equally suited for the learning task is rejected if a large fraction of the values is either positive or negative, and the probability of observing so many values of one type purely by chance is small. Section 3.1.1 describes the sign-test in detail. Table 3.1 describes the above two steps of ACR in pseudo-code. ACR needs five user supplied parameters. The parameter *start* is the first sample size considered by ACR, *finish* is the last sample size that is tried by ACR, *increment* is the size by which the sample size is increased, *max-runs* is the maximum number of runs after which it is decided that both representations are equally good, and *confidence* is the significance level at which the sign-test decides that one representation is better than another.

3.1.1 Ranking Representations with High Confidence

The idea of the sign test for ranking representations with the desired confidence is as follows. Suppose for the moment that ACR has been modified, so that if the minimum codelength for both the representations is the same in a run then it randomly chooses $(v1, v2)$ to be $(1, -1)$ or $(-1, 1)$ with equal probability. Now suppose that it were the case that no representation is better than the other, and any observed difference in the codelength is purely an artifact of the samples chosen for building a hypothesis. Then in a run $(v1 - v2)$ is equally like to be positive or negative. That is, each run is like tossing a coin where the probability of getting a head (positive value) or a tail (negative value) is equal to one-half.

Since runs are independent of each other, the probability of getting k or more positive values in n runs is given by:

$$\sum_{i=k}^{n} \binom{n}{i} 0.5^i 0.5^{n-i} = \sum_{i=k}^{n} \binom{n}{i} 0.5^n$$

If this probability is less than the desired confidence α, then we know that if it were the case that no representation is better than the other the probability of observing k or more positive values in n runs is less than α. Therefore, if we do observe k or more positive values in n runs we can reject the hypothesis that neither of the representations is better than the other and our chances of being wrong will be less than α.

For example, suppose that one wants to reject the hypothesis that no representation is better than the other with 99% confidence. Say that there are more positive values of $v1 - v2$ than negative values and this is the eleventh run[1]. One wants to determine the number k such that more than k positive values can occur in 11 runs only with probability less than 0.01. Let us try $k = 9$. The probability of getting

[1]The argument is symmetric if there are more negatives than positives.

Table 3.1 Pseudo-code for ACR.

procedure **rank** (*f1, f2, start, end, increment,max-runs, confidence*)
fi: the list of examples in *Ri*, i = 1,2.
start: the first sample-size, *end*: the last sample-size,
increment: increment for sample-size.
max-runs: maximum number of runs.
confidence: significance-level for **sign-test**.
positive ← 0, *negative* ← 0.
1. $R \leftarrow 1$.
2. do {
a. $(v1, v2) \leftarrow$ **run** (*f1, f2, start, end, increment*).
b. if $(v1 > v2)$ *positive* ← *positive* + 1
else if $(v2 > v1)$ *negative* ← *negative* + 1.
c. $S \leftarrow$ **sign-test**(*positive, negative, R, confidence*).
d. If $(S = 1)$, return($R1$ is better than $R2$)
else if $(S = -1)$, return ($R2$ is better than $R1$).
} while $(R \leq$ *max-runs*).
3. return (both representations are equally good).

procedure **run** (*f1, f2, start, end, increment*)
f1, f2, start, end, increment: same as in the procedure **rank**.
1. $S \leftarrow$ *start*.
2. do {
a. If the tests indicate that the codelength Ri is likely to decrease, then compute the codelength required to express fi with a hypothesis formed from a random sample of size S, $i = 1, 2$.
b. If the tests indicate that the codelength with $R1$ is unlikely to decrease, and the minimum codelength with $R1$ is greater than the minimum codelength with $R2$, then return (-1, 1).
c. If the tests indicate that the codelength with $R2$ is unlikely to decrease, and the minimum codelength with $R2$ is greater than the minimum codelength with $R1$, then return (1, -1).
d. $S \leftarrow S +$ *increment*.
} while $(S \leq$ *end*).
3. If the minimum codelength with $R2$ is greater than the minimum codelength with $R1$, then return (1, -1).
4. If the minimum codelength with $R1$ is greater than the minimum codelength with $R2$, then return (-1,1).
else return (0, 0).

more than 9 positive values in 11 independent runs is,

$$\sum_{i=9}^{11} \binom{11}{i} 0.5^{11} \approx 0.06$$

That is, if one observes 9 or more positive values in 11 runs one can reject the hypothesis that no representation is better than the other only with 94% confidence. Now try $k = 10$, in this case we see that probability of getting 10 or more positive values in 11 runs is less than 0.006. Therefore, if in eleven runs we do see ten or more positives we can be at least 99% confident that one representation is better than the other. In the situations where $v1$ and $v2$ are tied, we follow the usual practice of ignoring the tied values and reducing n accordingly (Gibbons, 1971).

3.1.2 A Test That Indicates Whether the Codelength Increases by Increasing the Sample-size

This section describes one of the two tests used for terminating a run. This test is based on the observation that if the learning algorithm is not able to generalize from the examples, then as samples of increasing size are presented to the learning algorithm the codelength required to describe the examples increases. If one assumes that there is no "critical" number of examples such that the ability of the learning algorithm to generalize improves dramatically for samples of size greater than this number, then a statistically significant trend indicating that the codelength increases with sample size can be expected to continue. If it can be established that the codelength increases with sample size, constructing hypotheses from larger samples will not help in reducing the codelength, and therefore one can terminate a run and report the minimum codelength in the run to be the minimum observed until it was established that codelength increases with sample size. The test described here serves as an indicator of the situation that codelength increases with sample size.

If a critical number of examples exists such that the ability of the learning algorithm increases dramatically for samples of size greater than this number, and this number is known, then the *start* parameter of ACR should be set to this number. This ensures that any indication about the codelength increasing with sample size is not based on unrepresentative sample sizes. Examples of learning algorithms for which such a critical number exists are various learning rules for a linear threshold unit (LTU) (Nilsson, 1965). The ability of LTUs to generalize changes dramatically once the sample size is greater than their capacity (Cover, 1965; Pearl, 1978). The capacity of a LTU is twice the number of variables present in the LTU. If the function is linearly separable, the accuracy of the hypothesis produced by the learning algorithm increases substantially. In general, for any learning algorithm that has a concept description language that describes a set of concepts with a finite VC-dimension, the accuracy of the hypotheses produced by the learning algorithm will increase substantially if the concept can be described in the concept description language, and the

number of examples is greater than a fixed function of the VC-dimension of the set concepts (Blumer, Ehrenfeucht, Haussler & Warmuth, 1989; Pearl, 1978).

ACR uses a non-parametric test of association between two variates, called *Kendall's rank test*, as an indicator of whether the codelength increases with sample-size (Gibbons, 1971; Kendall, 1962). The idea behind this test is that if there is no relationship between the number of examples and the codelength, then as the number of examples increases the codelength is equally likely to increase or decrease. However, if one observes that codelength usually increases with sample size, then one can determine the likelihood of getting such an observation by chance. To do this one records the number of times the codelength increases with sample-size. If this occurs in a large number of cases, then one determines the probability of getting so many cases of increase in codelength with increasing sample-size purely by chance, when in fact there is no relationship between codelength and sample size. If this probability is smaller than a desired confidence, 0.01 in our experiments, then the hypothesis of no relationship is rejected and the alternative hypothesis that codelength increases with sample size is accepted.

Specifically, let X denote the sample size and Y the codelength observed in a run for a sample of that size. Let $(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Yn)$ be a sequence of observations of codelength Y_i corresponding to sample of size X_i. If there is no relationship between codelength and sample size, then the codelength is equally likely to increase or decrease as sample size increases. That is, if we sum up the number of times codelength increases when sample size increases, and subtract from that the number of times codelength decreases when sample size increases we should get a number close to zero. A large positive number indicates that in most of the cases an increase in sample size results in an increase in codelength. Similarly, a large negative value indicates that an increase in sample size results in a decrease in the codelength.

If one defines

$$A_{ij} = \text{sign}(X_j - X_i)\text{sign}(Y_j - Y_i),$$

where,

$$\text{sign}(X) = \begin{cases} -1, & \text{if } X < 0 \\ 1, & \text{if } X > 0 \\ 0, & \text{if } X = 0, \end{cases}$$

then,

$$T = \sum \sum_{1 \le i < j \le n} \frac{A_{ij}}{\binom{n}{2}},$$

is a measure of how often codelength increases or decreases with sample size. Suppose codelength always increases when sample size increases; then in each of the $\binom{n}{2}$ pairs we have $A_{ij} = 1$ and $T = 1$. Similarly, if codelength always decreases when sample size increases then in each of the $\binom{n}{2}$ pairs we have $A_{ij} = -1$ and $T = -1$. The

ratio between $\sum\sum_{1\leq i<j\leq n} A_{ij}$ and $\binom{n}{2}$ is the measure of the relationship between X and Y. Kendell (1962) gives the formulas for computing the probability that a particular value of T is observed in n observations if there is no relationship between X and Y. Tables of these probabilities are also available (Hollander & Wolfe, 1973). Using these tables, one can determine the probability of obtaining a given value of T by chance when there is no relationship between sample size and codelength. If we find that T is positive, and that the probability of getting the observed value of T by chance is less than the desired value, we can reject the hypothesis that there is no relationship between codelength and sample size and accept the hypothesis that codelength increases with sample size. For example, for 99% confidence, we reject the hypothesis that there is no relationship between the sample-size and codelength if we observe a value of T that could have occurred only with probability less than 0.01 if the hypothesis of no relationship were true.

3.1.3 A Test That Indicates Whether the Codelength Can Decrease by Increasing the Sample-size

This section describes the second test for terminating a run. In the previous section we saw that an indicator of the situation that the instance representation is not suited for the learning algorithm is that codelength increases with sample size. Similarly, an indicator of the situation that the instance representation is well-suited for the learning algorithm is that codelength decreases with sample size. As more and more examples are presented to the learning algorithm it finds the regularities present in the examples, and this results in a compact hypothesis for the examples. However, after the learning algorithm has observed enough examples to extract all the regularities present in the examples, there are no more regularities present that can be used to produce a more compact hypothesis. Presenting more examples will not enable the learning algorithm to reduce the codelength further. One would like to be able to detect this situation and terminate a run. We have devised a test that, in the situations described below, indicates whether the codelength is likely to decrease by presenting additional examples to the learning algorithm. This test forms the basis for terminating a run.

To understand this test, let us look more closely at what we are trying to achieve with the two-part encoding. Suppose two people A and B are each given the same set of instances but only A knows the classification of the instances. In order to communicate the classification of the instances to B, it is required that A use as few bits as possible. A priori, A does not know which string needs to be transmitted, so he has to make provisions for transmitting any string. Therefore, A cannot reserve the shortest possible code for every string. For example, since only two strings can be described with one bit, to describe more than two strings A will need to reserve more than one bit for some strings.

Instead of trying to achieve the impossible goal of coding each string with the

fewest possible bits, A can minimize the average number of bits required to transmit a string. If A tries to achieve this goal, then in a number of situations if a string s occurs with probability $P(s)$, then the best A can do is to assign a code of length $-\log P(s)$ to s. In this sense, $-logP(s)$ is the ideal codelength for a string s. There are a number of proofs of this result, including the famous noiseless coding theorem by Shannon (1948).

To prove Shannon's theorem, assume that the coding scheme chosen by A is such that multiple strings can be encoded in such a manner that as multiple strings are transmitted, B knows when one string ends and the other begins. This is only possible if the code for a string is not a prefix of a code for any other string. Such codes are called *prefix-free* or *instantaneous codes* (Abramson, 1963). Let the length of the code for a string s in this scheme be $L(s)$. Appendix A gives a proof of Kraft's inequality that shows that for any prefix-free code $\sum_s 2^{-L(s)} \leq 1$. If in A's coding scheme $\sum_s 2^{-L(s)}$ does not add up to 1, then A can reduce the length of the code for at least one string so that $\sum_s 2^{-L(s)} = 1$. Assume, that A has done so and that no code word can be shortened without violating Kraft's inequality. Then, $Q(s) = 2^{-L(s)}$ is a set of numbers, one for each s, that adds up to 1. By the fact that if $\{Q(s)\}$ is any set of numbers adding up to 1, and $P(s)$ is the probability of s then,

$$-\sum_s P(s)logP(s) \leq -\sum_s P(s)logQ(s)$$

and the equality being achieved only if $Q(s) = P(s)$, we see that the shortest average codelength for a set of strings is achieved if each string is encoded with $-logP(s)$ bits. Most introductory texts on information theory prove this result (Abramson, 1963; Blahut, 1987). Thus, we see that if one is using a prefix code, then on average one cannot do better than assigning a code of length $-\log P(s)$ to a string s.

In fact, $-logP(s)$ is also the ideal codelength for a number of other coding procedures that are less restrictive than a prefix code. For instance, a *uniquely decodable* (UD) code is one where a concatenation of codewords for different strings can be uniquely decoded. Any set of codelengths achievable in a uniquely decodable code is also achievable in a prefix code (Abramson, 1963). So, the minimum expected codelength for a uniquely decodable code is also $-\sum_s P(s)\log P(s)$, which can be achieved only if every string is assigned a code of length $-\log P(s)$. Leung-Yan-Cheong & Cover (1978) have relaxed the condition of unique decodability and considered the minimal average codelength for *one-to-one* (1 : 1) codes. These are codes that assign a distinct binary codeword to each distinct outcome of a random-variable. There is no requirement that the concatenation of such codes be uniquely decodable. They show that the average codelength, $L_{1:1}$, of a one-to-one code for a random variable that can take N values is related to the average codelength, L_{UD}, of a uniquely decodable code by $L_{1:1} \geq L_{\mathrm{UD}} - loglogN - 3$. For strings of length n, $N = 2^n$, so we see from this relationship that L_{UD} is a first order approximation of $L_{1:1}$. Therefore, the minimal average codelength that can be achieved with a one-to-one code is roughly

equal to the average codelength that can be achieved with a uniquely decodable code, so $-\log P(s)$ can be taken to be the ideal codelength for a string s for a uniquely decodable code.

Moreover, Leung-Yan-Cheong & Cover (1978) also show that the average algorithmic information of a binary string s is also related to the entropy $H(S) = -\sum P(s)logP(s)$ of the set of S strings s. In particular, if $C(s|n)$ is the size of the shortest program for generating the binary string s of length n on a prefix machine, given the shortest program for computing n, and the fact that a computable distribution P governs the occurrence of the string s, then there exists a constant c such that $H(S) \leq E_p C(s|n) \leq H(S) + c$. $C(s|n)$ is known as the (conditional) Chaitin complexity of the string s (1975). The above shows that asymptotically in the Chaitin complexity of s, on average, given the shortest program to compute n on a prefix machine, the length of the shortest program to compute a string s of size n is given by $-\log P(s)$.

The above discussion shows that if we could determine the probability of occurrence of a string s, then we would know the minimum number of bits required to encode the string. Recall that the samples from which the hypotheses are produced are randomly selected, and if the learning algorithm is such that its hypothesis changes when the sample presented to it changes, then the occurrence of a particular error-list is a random event. Now suppose that during the course of taking random samples from the examples, an error-list s is produced for which one can determine the probability of occurrence $P(s)$. Can we say anything about the codelength of the consistent hypothesis? For instance, will adding more examples to the sample that produced the hypothesis that gave rise to s result in a consistent hypothesis with a shorter codelength? Since, $-\log P(s)$ is the ideal codelength for s, we know that, on average, if a new error-list is produced as a result of presenting additional examples to the learning algorithm, and the new error-list can be coded in k fewer bits than s, then at least k bits of changes must be made to the hypothesis. If we assume that these k bits of change increases the size of hypothesis by k bits, then we see that A cannot reduce the codelength by taking more examples. That is, once we can compute the probability of any string we know the minimum codelength that can be achieved for any consistent hypothesis. Below, we make this argument explicit.

By making two assumptions about the learning algorithm and the learning task, we can estimate the minimum number of bits required to describe the classification of the examples from the minimum number of bits required to describe an error-list. The minimum number of bits required to encode an error-list is given by the ideal codelength of the error-list. These assumptions are described below when they are required. The rationale for making these assumptions is discussed at the end of this section.

Say that a set of examples E produced a hypothesis h that resulted in the error-list for which the probability of occurrence can be determined. Call this error-list L, and let $S(m)$ denote the size of a hypothesis m. Suppose that e bits are needed

to describe the error-list. The question we ask ourselves is whether adding more examples to E would reduce the codelength required to describe the classification of the examples. Suppose it did. So now we have a new hypothesis, say h_1, with which the new error list, say L_1, can be described in, say, e_1 bits. The *first assumption* we make is that the learning algorithm is such that the size of the hypothesis does not get smaller by adding examples to a sample. Therefore, we know that the size of h_1 is greater than or equal to the size of h. The *second assumption* we make is that if adding examples to a sample changes the size of the hypothesis by r bits, then the new hypothesis can be produced from the old by specifying no more than r bits of changes. So, if it takes r bits more to describe h_1 than it takes to describe h, then h_1 can be produced by specifying no more than r bits of change to h. However, we have produced a contradiction. Specifying r bits of change and the error-list $L1$ to somebody who already knows h specifies a consistent hypothesis because this person now has h_1 and the corrections to h_1. From this consistent hypothesis this person can reconstruct L, because he already has h. Since we assumed that $S(h_1)+e_1 < S(h)+e$ and $S(h_1) = S(h) + r$, it must be the case that $r + e_1 < e$. That is we have encoded L with fewer bits than its ideal codelength.

So we see that, given our assumptions, if we could determine the probability of occurrence of an error-list, then we would know the minimum codelength of the consistent hypothesis for the example, and we can terminate a run. Unfortunately, we have no means of knowing the probability of occurrence of an arbitrary error-list. However, we can determine the probability of occurrence of certain error-lists. For example, if we know that a sequence of independent random events generates the string $s = x_1x_2 \ldots x_n$, and we also know the probability of occurrence of each event, then the probability of occurrence of s is the product of the probability of occurrence of the individual events.

Now suppose that in the course of taking samples of increasing sizes we get an error-list that is a sequence of independent random events. That is, we get an error-list where an error in a location i does not influence the probability of error in any other location. In other words, suppose that the presence or absence of any other example in the sample that produced the hypothesis (except of course the example corresponding to location i in the error-list), does not change the probability of error at the location i. Then, since the error-list is produced by a sequence of independent random events, we know the probability of occurrence of this string.

From the above discussion we know that if we could determine that the error-list was produced by a sequence of independent random events, then we could terminate a run. However, in general, testing for independence can be very expensive. It requires testing for each example e, whether hypotheses that change the classification of the other sets of examples change the classification of e. This would involve running the learning algorithm many times to produce the hypotheses that change the classification of other sets of examples and determining whether they change the classification of e. This is clearly an expensive proposition.

In ACR we use an approximate test for independence. Many approximate tests for independence are known, each test has some advantages and disadvantages (Knuth, 1971; Jansson, 1966). Different tests are based on different necessary conditions that must be satisfied by any sequence of independent random events. For implementing ACR we selected the test described below. Any other test that is effective in detecting independence can also be chosen for the implementation. Section 3.3 discusses an alternative test of independence, which could result in improved running time for ACR.

An approximate test for independence: The approximate test for independence, called the *serial correlation* test, is based on the fact that if the random variables marking the errors are independent, then knowing that there is an error in one location should not change the probability of error in the next location, or in a location separated by a distance of two, three, and so on. That is, if the random-variables marking the errors are independent, and the probability of making an error in any particular position i is p, then the probability of making the error at two positions i and j should be p^2. If on testing this pair-wise relationship between all pairs, we find that for random variables separated by some distance k, the average probability of occurrence of the error is different from p^2, then we know that these random variables are not independent. This is the idea of the *auto-correlation function.* It gives the correlation between random variables in the error-list separated by various distances, or *lags.* That is, if X_i denotes the random-variable marking the presence of an error at position i, μ is the mean of this random-variable, σ its variance, and $E(.)$ denotes the expectation operator, then:

$$r(k) = E(X_i X_{i+k}) = \begin{cases} \sigma^2 + \mu^2, & \text{if } k = 0 \\ \mu^2, & \text{if } k \neq 0 \end{cases}$$

The above equations assume that the mean and variance of random variables marking the errors are the same throughout the error-list, which we can assume because a hypothesis is formed by sampling uniformly from all the examples. The above equation can be converted into a more convenient form by subtracting μ^2 from each term, which gives:

$$s(k) = r(k) - \mu^2 = \begin{cases} \sigma^2 & \text{if } k = 0 \\ 0, & \text{if } k \neq 0 \end{cases}$$

The auto-correlation function only tests for pairwise linear correlations, and therefore, in general, it is possible for dependent random variables to have the above form for the auto-correlation function. This is the reason why it is an approximate test and not an exact test. However, the auto-correlation test is an exact test for independence if the random-variables being tested for independence are known to come from a normal distribution. But we do not assume that the random-variables marking the errors are distributed normally, and therefore the serial correlation test provides only an approximate test for independence.

To test for independence of finite length sequences, Chatfield (1984) recommends looking at the first few lags. He and Knuth (1971) also recommend considering an auto-correlation coefficient *significant* if it falls outside a desired confidence interval around zero, and considering a sequence of random variables independent if the number of significant coefficients is below a threshold determined by the confidence interval. The confidence intervals have been computed based on exact calculations for some distributions and sampling experiments for many other distributions. In our experiments, one fourth of the coefficients were analyzed and the error-list was considered independent if less than 5% of the coefficients fell outside a 95% confidence interval around zero. The results presented in the next chapter indicate that the approximations, made in order to estimate quickly whether an error-list is independent, are good enough to permit the solution of the representation comparison problem in many situations.

The rationale for the assumptions required for the independence test: We do not know whether the two assumptions made for terminating a run are necessary for designing effective indicators of whether the codelength is likely to decrease once we estimate the probability of occurrence of a particular error-list. These assumptions are based on our experience with algorithms that build decision trees and perceptron trees. For these algorithms we have observed that complexity of the hypotheses produced by the learning algorithm usually increases with sample size. In fact, this is usually true for algorithms that generalize by producing a compact hypothesis consistent with the examples. This is because the examples not present in any subset of a sample can be set to any value that helps in making the hypothesis compact. However, when these examples are present in the sample they add additional constraints that make a consistent hypothesis more complex.

When B has no hypothesis, to specify a hypothesis of size r bits A has to transmit r bits to B. The second assumption is that this property holds no matter what hypothesis H is currently held by B. If by adding examples H changes by r bits, then A needs to specify no more than r bits of changes to produce the new hypothesis from H. This assumption formalizes the idea that modifying a hypothesis by incorporating new instances does not cause a major restructuring of the hypothesis. That is, if incorporating new examples produces a new hypothesis that differs from the original hypothesis by a small amount then only the changes to the old hypothesis need be specified. Most of the original hypothesis remains unchanged. This will be true if the hypothesis consists of a number of parts and adding new examples causes the hypothesis to change in only a few parts, the other parts remain unchanged. For instance, if the hypothesis is a decision tree then adding more examples causes only a few branches of the tree to change and most of the tree remains unchanged. Similarly, if the hypothesis is a boolean formula, then only a few terms in the formula change and most terms remain unaffected. Notice that this assumption does not preclude the situation that a large change in the hypothesis may be caused by specifying a small change. For instance a specification like, "incorporate the instance e into the

current hypothesis" may produce a large change in the hypothesis, but this large change can be specified by a small number of bits. The only situation that the assumption precludes is that the size of the hypothesis changes by a small amount, but it takes a large number of bits to specify the change. This can occur if there is a major restructuring of the hypothesis without a substantial change in the size of the hypothesis. Finally, even though we require that this assumption be true at all times, ACR makes use of this assumption only after an independent error-list has been obtained. Once an independent error-list is obtained, there are no regularities left in the examples, so the learning algorithm does generalize from a new example, causing only a change to the part of the hypothesis that applied to the example.

One may wonder about the sensitivity of ACR to the above assumptions, and about the effectiveness of the heuristic test for independence. The results presented in the next chapter show that for two learning algorithms, namely ID3 and PT1, ACR is effective in ranking the representations correctly for many problems. We take these results as an empirical demonstration of ACR's effectiveness in ranking representations. The assumptions required for the two tests describe the ideal conditions that form the basis of ACR. Practical learning algorithms are approximations of these ideal conditions and the empirical results show ACR's robustness in dealing with these approximations.

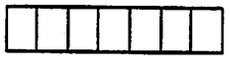
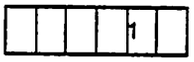
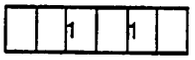
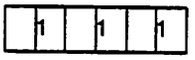
3.2 An Illustrative Example

To illustrate the steps involved in ACR, this subsection walks through the algorithm using two representations of the *two-clumps* problem. In this problem the task is to learn to recognize whether there are exactly two contiguous blocks of black pixels in a one-dimensional visual field. This problem belongs to a class of problems devised by Denker et. al. (1987) to illustrate the effect of instance representation on learning. There are two representations for the visual field. The first, called the *Pixel* representation, tells whether each pixel is black or white. The second, called the *Edge* representation, records a 1 at every white pixel where there is a transition from a black pixel to a white pixel, and records a 0 otherwise. A 1 denotes an edge between black and white pixels. Edges are defined at the boundary by assuming that the visual field is embedded between two white pixels. Table 3.2 gives different representations of a seven-pixel-wide visual field, two of these pixels that form the border are always white. In the Edge representation 0's are not shown for clarity.

We will illustrate ACR with the two-clumps problem for a nine-pixel-wide visual field, two of these pixels that form the border are always white. This is a small problem that nonetheless demonstrates all aspects of ACR. The examples are generated for the Pixel representation in a lexicographic manner, and for every example in the Pixel representation the corresponding example in the Edge representation is determined.

A sample of 20 instances is selected randomly, with replacement, from the set of

Table 3.2 Alternative representations of the visual field for the clumps problems.

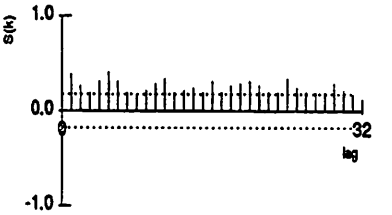
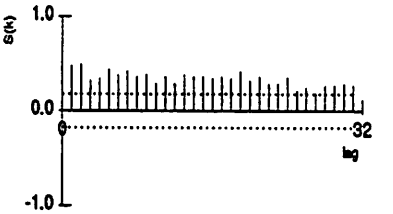
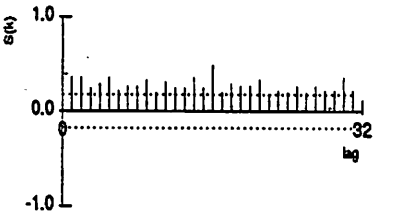
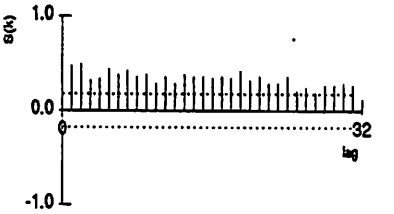
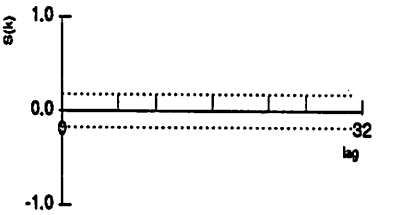
PIXEL	EDGE	Meaning
		0 clumps
		1 clump
		2 clumps
		3 clumps

examples in the Pixel and Edge representations and presented to ID3. Each example has equal probability of being selected. The decision trees produced by ID3 are used to classify the 128 examples in the respective representations. The first row of Table 3.3 shows the first 40 bits of the error-lists that result from these trees, along with the auto-correlation for the first 32 lags.

The dotted lines shows the interval within which 95% percent of auto-correlation coefficients should lie if the random variables marking the errors are independent. The fourth column of this table shows the codelength of the resulting consistent hypothesis. A method for encoding decision trees similar to Quinlan and Rivest's (1989) method is used to compute number of bits required to describe a decision tree. We have modified this method so that the code for a decision tree forms a prefix-free code. Cover's (1973) enumerative coding method gives the codelength of the error-list. Section 4.1.2 gives the exact procedures used for determining the codelengths.

The first row of Table 3.3 shows that more than 5% of the auto-correlation coefficients are outside the desired interval for both the representations, therefore ACR concludes that errors are not independent for both the representations. Also, the trend between codelength and sample size has not been observed sufficiently enough to conclude that the codelength increases with sample size. Now, a sample of 30 instances is taken and the above procedure repeated. The second row of Table 3.3 shows the error-list and the auto-correlation for this sample. The conclusion is the same as above, nothing can be said yet about the minimum codelength. When the sample size becomes 60, ACR observes a new minimum of 180.49 bits for the codelength with the Edge representation. When the sample size becomes 110, for the Edge representation there are less than 5% (actually none here) auto-correlation coefficients above the desired confidence interval. The codelength for the consistent hypothesis that results from this sample is 187.73 bits. ACR concludes that adding

Table 3.3 A sample run of ACR

Sample Size	Error vector (First 40 bits)	Auto-correlation	Codelength
20 (E)	1100101000010000000011100000000000010001		209.30
20 (P)	1111000100000001000011100000000110100001		189.22
30 (E)	0110101000100001000011100000000100001110		208.58
30 (P)	0010011011000101110010001100010111011001		224.37
110 (E)	0000000000000000000000000000000000000001		187.37
120 (P)	0000000000000000110100100000000000000100		351.78

Note: E denotes the Edge representation and P denotes the Pixel representation

more examples to this sample will not decrease the codelength further and based on this run the least value of the codelength seen so far— 180.49 bits— is an estimate of the minimum codelength with the Edge representation.

A lower bound on the codelength with the Pixel representation has still not been estimated. The sample size is increased to 120 only for the Pixel representation, and one gets a consistent hypothesis with a codelength of 346.78 bits. Now one finds that 11 observations of the codelength have been made since the last time a new minimum was observed for the Pixel representation. In these $\binom{11}{2} = 55$ pairs, 37 times the codelength increased with sample size. For 99% confidence in concluding that codelength increases with sample size one needs to observe that codelength increases at least 31 out of 55 times (Hollander & Wolfe, 1973). ACR therefore concludes that for the Pixel representation the codelength increases with sample-size, and 189.22 bits is an estimate of the minimum codelength. This is greater than the estimate of 180.49 bits for the Edge representation, and hence from this run ACR concludes that for the nine-pixel-wide two-clumps problem, the Edge representation is better than the Pixel representation. To ensure that the samples drawn in this run are not unrepresentative, 10 more runs are required to establish with 99% confidence that the Edge representation is better for this problem.

To obtain a ranking of the representations based on estimating the error of the hypothesis produced with different representations of this problem, one would need to estimate the accuracy of the classifier produced with different sample sizes. Also, one will need to choose a method for estimating the accuracies of the various classifiers. Due to the small number of examples, an expensive resampling method will be needed for reliably estimating the accuracies. The savings obtained by ACR may not appear significant in this illustrative example, but as the next chapter shows, as the problems become larger the savings obtained by using ACR become increasingly significant, even for an inexpensive (though potentially inaccurate) method of accuracy estimation.

3.3 Extensions to the Implementation

The serial correlation test is one of a number of tests that can be used for detecting independence (Knuth, 1971; Jansson, 1966). Another test that can be used is the *frequency test.* In this test one would divide the error-list into a number of non-overlapping regions, say three regions A, B, and C. If the errors are independent then one expects that the frequency with which an error occurs in the region A should be the same as the frequency in the regions B and C. Therefore, one can determine the frequency of occurrence of the errors in the region A and check to see whether the same frequency is observed in the region formed by combining B and C. As shown in Knuth (1971), one can derive the acceptable regions for the discrepancy between the observed and expected frequencies. The acceptable region will depend

on the confidence with which one wants to establish that the errors are independent. If the number of errors in the region formed by combining B and C fall within the acceptable regions, then the random-variables marking the errors are considered to be independent. The test is conducted two more times, once by using B to establish the expectation, and testing it against the combination of A and C, and once by using C to form the expectation and testing it against the combination of A and B.

The reason we did not use this test was because it tests the global properties of a sequence and it may fail to detect short range periodicities. On the other hand, the serial correlation test will tend to detect short range periodicities when the lags are small, and longer range periodicities for larger lags. By ensuring that correlations at all lags fall within the acceptable range, the serial correlation test checks for the presence of both these types of regularities. However, it has been our experience that the serial correlation test is a very conservative test for determining whether the codelength is likely to decrease with increasing sample-size. Often, the codelength stops decreasing but the serial correlation test does not consider the errors to be independent. A few sample-sizes later, the serial correlation test does find the errors to be independent. It is possible that the frequency test, or some other test for independence, will detect that the codelength is unlikely to decrease earlier than the serial correlation test. Also, since the frequency test requires less computation time, the use of the frequency test may reduce the running time of ACR. Instead of $O(N \log N)$ time required to compute the auto-correlation of N quantities, the frequency test can be conducted in $O(N)$ time. Another method for making ACR faster would be to conduct the serial correlation tests only for sequences that pass the frequency test. Though this will not reduce the worst case complexity, it will reduce the expected running time because sequences that are obviously not random will fail the frequency test.

The current implementation of ACR ranks representations only for problems where the task is to learn the classification of a set of instances. The extension to learning integer and real-valued functions is straight-forward. The error-list in these cases will not be a bit-string but a sequence of numbers. In the case that these numbers are integers, a prefix code for integers can be used to determine the number of bits required to encode the error-list (Elias, 1975). In the case that these numbers are real-valued, one can choose some precision for encoding real-valued numbers, change the real-valued numbers to integers by multiplying by the precision, and use a prefix code for integers. The rest of the algorithm does not change.

The current implementation compares only two representations at a time. The best in a set is found by making pairwise comparisons, at each time comparing one representation with the best so far. A straight-forward extension to find the best representation in a set simultaneously would be to set up a table with an entry for each pair. Whenever it is estimated that the codelength with a representation R is unlikely to decrease, an entry is made in all the pairs that involve R. If it becomes established with the desired confidence that R is worse than some other representation, then all

pairs involving R are removed from future consideration. This provides a savings in the running time because the minimum codelength achievable with a particular representation need not be estimated repeatedly.

Choosing the parameters for ACR: All the experiments reported in the next chapter were conducted with the same parameter settings as those used in the example presented in Section 3.2. The significance level for the sign-test was set at 99%; and if no representation was found to be better than the other with this significance level within 25 runs, then both the representations being compared were considered to be equal. The significance level for the rank test was set at 99%. For the independence test, 95% confidence intervals were used to determine the number of significant correlations.

In a specific application of ACR, one might want to use different settings for these parameters. For instance, often fewer runs are required to rank representations if the significance level for the sign-test is reduced to 95%. This is useful when the difference in the codelengths is small and a large number of runs are required to rank representations with 99% confidence. This happens when one representation is only slightly better than the other. On the other hand, if it is necessary to have high confidence in the ranking then might want to consider increasing the maximum number of runs within which ACR has to rank the representations.

Similarly, one might want to consider reducing the significance level for the rank test to 95%. This will provide an early indication of the situation where the codelength increases with sample size, consequently time required for a run will be reduced. However, there is an increased risk of considering situations where codelength does not increase with sample size as being those where codelength increases with sample size. Finally, as mentioned above, we have found the auto-correlation test to be conservative in declaring that the error-list is independent. Often, the codelength stops decreasing a few sample sizes before the error-list passes the independence test. One method for obtaining a quicker test for determining whether the codelength is likely to decrease with increasing sample size, is to lower the confidence interval for considering an correlation significant.

The exact settings for parameters will depend on the particular application of ACR, and on the confidence desired in the ranking. The settings used in our experiments were found to be effective for a variety of problems. In a new application, one can start with these settings and change them if ACR cannot rank the representations in 25 runs. If in a large fraction of these runs ACR does find one representation to be better than the others, then a small increase in the number of runs may produce a ranking with the desired confidence. If this does not happen, then the significance level for the sign-test can be reduced. If this too does not produce a ranking, then the representations are probably equally suited for the learning algorithm.

3.4 The Worst-case Complexity of ACR

This section analyzes the worst-case running time of ACR. The worst case for ACR occurs when both representations being compared are equally good and ACR cannot estimate the minimum codelength by presenting samples of small sizes to the learning algorithm. Usually ACR takes less time than the worst case; the worst case has never occurred in our experiments.

Let N be the number of examples available to ACR; S be the maximum number of sample points in a run; and $L(N)$ be the time taken by the learning algorithm to form a hypothesis with N examples and to classify the N examples. $L(N)$ is an upper bound on the time taken by a single call of ACR to the learning algorithm and the time taken to obtain the error-list.

In a run ACR makes at most S calls to the learning algorithm. For each call of the learning algorithm, it takes an additional $O(N \log N)$ steps to compute the auto-correlation function using the fast Fourier transform. For the rank test the number of times the codelength increases with sample size is computed by maintaining a sorted list of codelengths. This requires at most S steps, and since $S \leq N$, an upper bound on the time taken for the two tests is $O(N \log N)$. Putting all the above steps together, an upper bound on the time taken for a run is:

$$O(S[L(N) + N \log N]).$$

If R is the maximum number of runs allowed— within which if no representation is considered to be better than another with the desired confidence, ACR concludes that both representations are equal— then the worst-case cost of ranking two representations with ACR is:

$$O(RS[L(N) + N \log N]).$$

In the experiments reported in the next chapter, two representations were considered equal if within 25 runs ACR did not find one to be better than another at the 99% significance level. The number of runs required in any particular case depends on whether the representations are well-suited for the task and how much better one representation is compared to the other. Usually, if one representation is better than another seven runs suffice for determining this at the 99% significance level. A minimum of seven runs are required because with less than seven samples one cannot have 99% confidence via the sign test.

3.5 Summary of the Chapter

This chapter presented the details of ACR, an algorithm to compare representations. ACR ranks representations by estimating the minimum number of bits required to specify a consistent hypothesis for the examples in the different representations. To estimate the minimum codelength efficiently, ACR observes the trend

in the codelength as an increasing number of examples are presented to the learning algorithm. After observing a sample from the examples, ACR conducts tests that indicate whether the codelength is likely to increase with sample size and whether the codelength is likely to decrease. If the rank-test indicates that the codelength increases with sample size, then ACR concludes that with high likelihood presenting more examples to the learning algorithm will not reduce the codelength. Similarly, if the independence-test indicates that with high likelihood the codelength will not decrease further, then ACR stops presenting examples to the learning algorithm. Representations are ranked based on the estimated size of the minimum codelength for different representations.

Section 3.1.2 discussed the rank test that indicates whether the codelength increases with sample size. This test is based on the fact that if the codelength does not increase with sample size, then by increasing the sample size the codelength is equally likely to increase or decrease. If ACR observes that in a large number of cases increasing the sample size increases the codelength, it determines the probability of this event occurring by chance and concludes that codelength increases with sample size if this probability is small. Section 3.1.3 discussed the independence-test that indicates whether the codelength is likely to decrease if additional examples are presented to the learning algorithm. This test is based on the fact that if knowing the classification of one example tells nothing about the classification of the other examples, then there is no pattern left in the examples that can be utilized to reduce the codelength. ACR used an approximate test of independence to establish that knowing the classification of one example tells nothing about the classification of other examples. Finally, extensions to the current implementation of ACR were discussed and the worst-case complexity of ACR was analyzed.

CHAPTER 4

TESTING ACR'S PERFORMANCE

This chapter describes the results obtained by using ACR to rank alternative instance representations for a number of tasks. The experiments reported here had two objectives. The first objective was to determine whether ACR correctly ranks the representations. The second objective was to determine whether it was faster to rank representations by using ACR rather than directly estimating the accuracy of the hypotheses produced by the learning algorithm with the different representations.

To meet these objectives, ACR was used to rank some representations for the tasks described below. ACR's ranking and the time taken to obtain the ranking was recorded. To verify ACR's predictions, the accuracy of the hypotheses produced by the learning algorithm with the different instance representations was estimated by two methods. The first method, called the *hold-out* method, took two disjoint sets of examples called the training and the testing sets[1]. The accuracy of a hypothesis produced from a sample of examples from the training set was estimated by determining the accuracy on the testing set. The sample that enables the learning algorithm to produce the most accurate hypothesis is not known a priori, therefore the accuracy was determined for samples from the training sets of different sizes. The time required to estimate the accuracy of the hypotheses produced with different representations was recorded. The reason for choosing the hold-out method was that it is an inexpensive method for estimating the accuracy of a hypothesis. Therefore, if ACR can rank representations faster than the ranking obtained by using the hold-out method, then we know that it is faster to rank representations by using ACR than by directly estimating the accuracy of the hypotheses using methods of estminating accuracy that are more expensive than the hold-out method.

During the course of our experiments with the hold-out method, we found that for some tasks changing the testing and the training sets changed the conclusion about the better representation. For the tasks where the difference in the accuracy with the different representations was small, a second set of experiments was conducted with a more expensive accuracy estimation procedure, namely *ten-fold cross-validation.* In this procedure the examples are divided into ten different training and testing sets, and the hold-out procedure is conducted for each of these ten different partitions.

[1]Strictly speaking, these are multi-sets because examples can be repeated.

ACR's effectiveness in ranking representations was evaluated as follows. On the tasks where the ranking based on a single testing set did not change with different hold-out sets, ACR's ranking was considered to be correct if it matched the ranking obtained by using the hold-out method. For the tasks where the ranking obtained from a single hold-out set changed when the hold-out sets were changed, ACR's ranking were compared with the conclusions that could be drawn by using cross-validation to rank representations.

These experiments show that ACR is effective in ranking representations for a variety of tasks. These tasks included simple perceptual tasks used in the literature to illustrate the importance of instance reprsentations for learning, tasks involving the simple board game of Tic-Tac-Toe, and tasks involving learning to recognize user-independent handwritten characters. In addition, it took less time to rank representations of the above tasks by using ACR than directly estimating the accuracy of the hypotheses produced by the learning algorithm. The rest of this chapter describes the experiments and results that support this conclusion.

4.1 Details of the Experiments

The experimental strategy used to establish ACR's effectiveness was the following.

1. For each task, the examples in each representation were divided without replacement into two sets. Ninety percent of the examples formed the training set and the remaining ten percent of the examples formed the testing set.

2. For each task, only the examples in the training sets were given to ACR. ACR's ranking of the representations was recorded. The time required to obtain the ranking was recorded as ACR's time [2].

3. For each task, the validity of ACR's prediction was verified by estimating the error-rates of the hypotheses produced by the learning algorithms from samples of various sizes. Error-rates were estimated as follows. For each representation samples of increasing size were drawn with replacement from the respective training sets. These examples were presented to the learning algorithm. The fraction of the examples in the test set correctly classified by the classifier produced by the learning algorithm was recorded as the accuracy of the classifier. Multiple runs were conducted for each sample size until it was established at the 99% significance level via the sign test that there is a difference in the accuracies of the classifiers produced with the different representations. However, if after 100 samples the differences were still not significant, the average value of

[2] All experiments were conducted on a SUN-4. ACR and error estimation procedures share the code for the learning algorithm, input-output and the significance test.

accuracy in the 100 samples was recorded. The values for the smallest sample size, the sample size increment and the maximum sample size were identical to the values used by ACR. Appendix D lists these values. The time taken for this procedure was recorded as the time for the hold-out method.

4. ACR's prediction was considered correct if, over all sample sizes tested, the representation predicted to be the best by ACR produced the hypothesis with the highest accuracy.

4.1.1 The Learning Algorithms

ACR was used to rank representations for the following learning algorithms.

ID3: an algorithm to build decisions trees. ID3 is a popular algorithm for constructing decision trees (Quinlan, 1983; Quinlan, 1986). In addition to machine learning, decision trees are also used for non-parametric regression (Breiman, Friedman, Olshen & Stone, 1984). ID3 uses an information theoretic heuristic to build a small decision tree for the observed examples. The main idea of the algorithm is to partition the examples into subsets based on the value of one of the attributes so that the "ambiguity" in the classification of the examples is minimized. This process is continued recursively for each partition until there is no ambiguity in the classification of the examples. In our experiments the information-gain criterion was used for attribute selection and the trees were built without pruning.

PT1: an algorithm to build perceptron trees. Perceptron trees are decision trees where the leaves are allowed to be linear threshold units (LTU) (Utgoff, 1989). PT1 tries to find a LTU at each node that correctly classifies all instances present at that node. However, a LTU cannot correctly classify a set of non linearly separable instances. PT1 determines heuristically whether the instances at a node can be separated by a hyperplane. If the instances cannot be separated by a hyperplane, PT1 partitions the examples into subsets based on the value of an attribute, just as in ID3. This process is continued until either the examples at a node are separable by a hyperplane or there is no ambiguity left in the classification of the examples. Since PT1 is not guaranteed to produce a perceptron tree consistent with all the examples in the sample in one pass, examples in the sample were presented repeatedly to PT1 until it produced a tree consistent with all the examples.

4.1.2 Codelength Formulas

To determine the number of bits required to encode a consistent hypothesis, the number of bits required to encode an error-list and the number of bits required to encode the hypothesis produced by the learning algorithm are needed. The coding method used here for determining the number of bits required to encode an error-list is general and it can be used with any learning algorithm. However, for each new learning algorithm a method for determining the number of bits required to encode

a hypothesis needs to be designed to ensure that it correctly reflects the bias of the learning algorithm.

For instance, if the learning algorithm searches the space of hypotheses in the order of increasing complexity, then short encodings should be reserved for low complexity hypotheses. For example, since ID3 has a bias to produce small trees, for ID3 coding of decision trees must assign short codes to small trees. If on the other hand, the learning algorithm that has a bias to produce the largest decision tree consistent with the examples, the encoding method should assign short codes to large trees.

Another way to look at the constraints imposed by the bias of the learning algorithm on the encoding scheme for hypotheses is to consider the encoding scheme as a method for assigning prior probabilities to various hypotheses. If a prefix code is used for encoding the hypothesis, then by Kraft's inequality a hypothesis of size l can be assigned a prior probability of 2^{-l}. Given this assignment of prior probabilities, learning from a set of examples by using the minimum description principle can be viewed as choosing the hypothesis with the maximum a posteriori probability.

The bias of a learning algorithm reflects the prior probability assigned by the learning algorithm to various hypotheses. Therefore, the encoding scheme should be such that it assigns higher probability to the hypotheses considered a priori to be more likely by the learning algorithm. That is, since 2^{-l} decreases with increasing l, the encoding scheme should assign short codes to hypotheses that are assigned higher â-priori probability by the learning algorithm. Another consideration while designing the encoding scheme is that it should contain all the information required to reconstruct the hypothesis. Only the information that is common to *all* hypotheses can be excluded from the encoding procedures. Also, various components of the hypotheses should be encoded as compactly as possible. Since the description length of the hypothesis is involved in a trade-off with the description length of the error-list, the encoding scheme must not favor any of the two parts involved in the trade-off.

From the above remarks it may appear that designing a method for determining the description length of a hypothesis requires considerable skill. However, our experience has been that it was possible to design simple encoding schemes without much effort. These simple encoding schemes were refined as ACR was being developed, but the ranking produced by ACR were robust against these refinements. The encoding schemes used in our experiments are described below. Different parts of a consistent hypothesis were encoded with a prefix code, so the different parts can be concatenated and the receiver can distinguish them from each other to construct a consistent hypothesis.

Coding Integers: A single integer was encoded using Elias's (1975) asymptotically optimal prefix code for integers. This code begins with a zero, and writes the binary representation for the integer to the left of the zero. Say the length of the binary representation is l. The binary representation of l is then prefixed to the code. The binary representation of the length of the code for l is then prefixed to the code, and so on, until the length of the last binary representation written becomes 1 bit.

Table 4.1 Asymptotically optimal Elias code for integers

Integer	Code
1	0
2	10 0
3	11 0
4	10 100 0
7	10 111 0
8	11 1000 0
15	11 1111 0

Table 4.2 Computing the codelength of the Elias code for the integer j.

1)	length $\leftarrow$ 1.
2)	if $\lfloor \log j \rfloor = 0$, return length.
3)	increment length by $\lfloor \log j \rfloor$.
4)	j $\leftarrow \lfloor \log j \rfloor$.
5)	go to 2.

Table 4.1 gives the Elias code for some integers. For readability the various parts of the code are separated by blanks. Table 4.2 gives an algorithm to compute the length of the Elias code.

Coding the error-list: The error-list is encoded by using Cover's (1973) enumerative coding scheme. Suppose we know the length of the error-list, say it is n, and we know that the error-list contains m 1's. Then any one of the $\binom{n}{m}$ error-lists can be specified with $\log \binom{n}{m}$ bits. Denote this length by $T(n,m)$. As shown by Risassnen (1989), we can approximate $T(n,m)$ by,

$$nH(m/n) - \frac{1}{2}\log\frac{m(n-m)}{n},$$

where $H(p) = -p\log p - (1-p)\log(1-p)$ is the binary entropy function. However, since m is not known to the receiver, we first transmit a prefix code for the integer m followed by $T(n,m)$ bits for the error-list. Independent error-lists are encoded by assigning to them a code of length given by the ideal codelength $-\log P(s)$. For classification tasks where there are more than two classes, $T(n,m)$ must be multiplied by the binary logarithm of the number of classes because in addition to the location of the errors, the correction also has to be specified.

Coding decision trees: To encode a decision tree, we can traverse the tree and for each node specify:

1. whether it is a decision node or a leaf.
2. if it is a decision node, the attribute tested at that node.
3. if it a leaf, the class represented by the leaf.

This encoding assumes that the attributes of each node are in an order known to the receiver, and the branches come out of a node in this order. For instance, if an attribute called `color` has three values `red`, `green`, and `blue`, the receiver on receiving this attribute will know that the left most branch coming out of this node is traversed when `color = red`, the middle branch is traversed when `color = green`, and the rightmost branch is traversed when `color = blue`.

If an internal node is represented by a zero and the leaves are represented by a 1, we can use the method used for encoding the error-list to encode the structure of the tree. To do this we need to transmit to the receiver the following information:

(a) The sum of the number of leaves and the number of internal nodes.

(b) The number of internal nodes.

(c) The code for the string formed by using a zero to mark the leaves and 1 to mark the internal nodes.

Part (a) and (b) can be encoded by using a prefix code for integers, and part (c) by the enumerative code used for encoding an error-list. This scheme for encoding the structure of a decision tree is similar to the one used by Quinlan and Rivest (Quinlan, 1989). However, there are two differences. First, we compute the length of prefix code for transmitting the number of positions that have a one in the error-list. Quinlan and Rivest suggest using a fixed-length code of size given by the logarithm of the length of the error-vector, because this number is an upper bound on the number of bits required to specify any number for the errors. The second difference is that to encode the structure of the tree Quinlan and Rivest do not transmit the part (a) above. As a result, the receiver does not know how many bits of the structure should be expected, and where the structure ends and other things being transmitted begin. By transmitting this number we ensure that there is no ambiguity of this type in the coding process. Furthermore, the code for the decision tree forms a prefix code, so the receiver knows where the code for the decision tree ends.

If k denotes the number of classes, then the class represented by a leaf can be encoded with $\log k$ bits. If there are n leaves the number of bits needed to encode the classes is $n \log k$. To encode the attribute names at each of the internal nodes, the depth at which the node occurs is required. Say there are r attributes. The root of a decision tree could be any one of the r attributes so $\log r$ bits are needed to

specify the attribute at the root. At depth one in the tree only one of $r-1$ remaining attributes are allowed, so $\log(r-1)$ bits suffice. In general, if d_i denotes the number of nodes at depth i, then the internal nodes can be encoded with a number of bits given by:

$$\sum_{i=0}^{r-1} d_i \log(r-i).$$

Rissanen (1989) gives a simliar coding scheme. Our scheme differs from his because we do not assume that the tree always begins with an internal node and ends with a leaf. A single-node tree may not have this property. By assuming that a tree always begins with an internal node and ends with a leaf, part (c) of the code for the structure of the tree can be reduced by not transmitting the first and the last bit.

Coding perceptron trees: The encoding of a perceptron tree is very similar to the encoding used for a decision tree, except that an additional leaf type, called perceptron, is added. If n is the number of leaves (including perceptrons), and there are k classes, then the number of bits required to encode the leaves is $n\log(k+1)$, the one has been added to account for the additional leaf type. The weights of the perceptron are encoded by reserving one bit for the sign and encoding the magnitude of the weight with the above prefix code for integers.

An alternative method for encoding a perceptron could be to assume fixed precision weights and to encode a perceptron by multiplying the number of weights with the number of bits required to encode each weight. However, the weights of the perceptrons in a perceptron tree start from small values and grow in response to the corrections. A small weight indicates that the learning algorithm did not have to consider a large number of hypotheses along the dimension represented by that weight. Similarly, a large weight occurs in response to a large number of corrections, indicating that a large number of hypotheses were explored along that dimension. Therefore, we consider that an encoding in which the length of the code increases with the magnitude of the weight captures the bias of the perceptron tree algorithm better than fixed size weights.

4.1.3 The Learning Tasks

ACR was used to rank representations of the following tasks.

The Clumps problems

This class of problems was used in the illustrative example presented in Section 3.2. In these problems, the task is to learn to recognize various patterns in a one-dimensional visual field. ACR was used to rank the Pixel and the Edge representations of a thirteen-pixel-wide one-dimensional visual field. In the Pixel representation, the visual field is described by giving the color of each pixel. In the Edge representation, the visual field is described by marking the transitions from a black pixel to a white pixel by a 1 and all other transitions by a 0. The visual field is assumed to be embedded between two white pixels.

Table 4.3 Ranking the instance representations of the Clumps tasks for ID3.

Problem name	ACR's prediction	ACR's savings (CPU secs.)
Two-or-more-clumps	Edge better than Pixel	$\approx$ 4 times
Two-clumps	Edge better than Pixel	$\approx$ 29 times

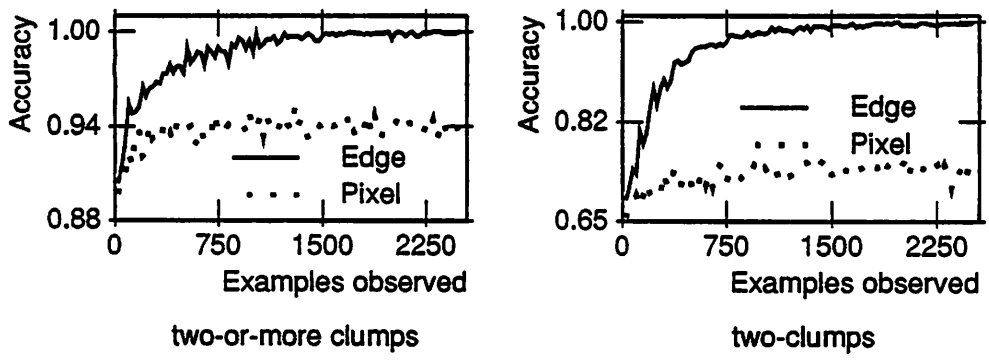


Figure 4.1 The accuracy of ID3 on test sets for the Clumps problems.

The Pixel and the Edge representation were compared for the following two problems in this class:

1. The two clumps problem. The task is to learn to detect the presence of *exactly two* contiguous blocks of black pixels.

2. The two-or-more-clumps problem. The task is to learn to detect the presence of two or more contiguous blocks of black pixels.

Table 4.3 gives ACR's ranking of the Pixel and the Edge representations for ID3. For both the problems, ACR predicts that the Edge representation is better than the Pixel representation. The third column of Table 4.3 gives the ratio of the time taken by the hold-out method to the time taken by ACR. For the two-or-more-clumps task ACR took 2919.13 CPU seconds to rank the representations and the hold-out method took 10961.06 CPU seconds, giving a savings of roughly 4 times. For the two-clumps problem, ACR took 598.95 CPU seconds and the hold-out method took 17611.08 CPU seconds, giving a savings of roughly 29 times. Figure 4.1 shows the accuracy of the decision trees produced by ID3 for these two tasks from samples of different sizes. This figure shows that ACR correctly ranks the representations of the Clumps task for ID3. In addition, ACR ranked representations with savings in computation time ranging from 4 to 29 times over the hold-out method.

Table 4.4 shows the ranking produced by ACR for PT1. Figure 4.2 gives the corresponding graphs for the estimated error-rates. The conclusion of the experiments

Table 4.4 Ranking the instance representations of the Clumps tasks for PT1.

Problem	ACR's prediction	ACR's savings (CPU secs.)
Two-or-more-clumps	Edge better than Pixel	$\approx$ 23 times
Two-clumps	Edge better than Pixel	$\approx$ 70 times

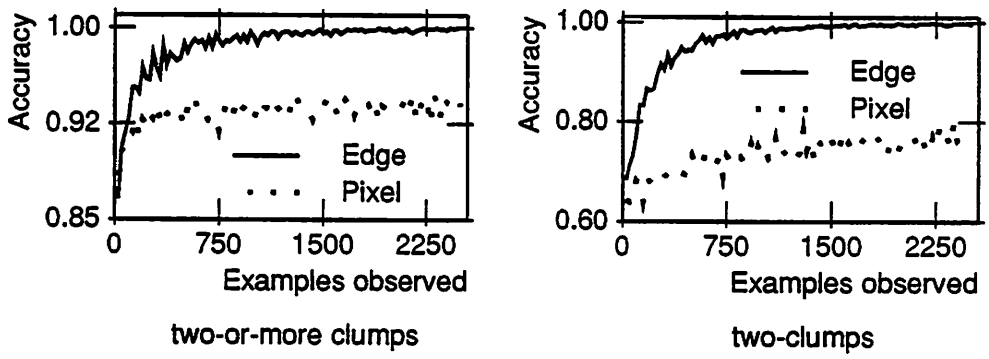


Figure 4.2 Accuracy of PT1 on test sets for the Clumps problems.

with PT1 is the same as for ID3, ACR correctly ranks the representations of the Clumps task for PT1. In addition, it is faster to rank representations by using ACR than by estimating error-rates. Notice that the savings obtained by using ACR are much more for PT1 than ID3. This is because the PT1 algorithm usually takes more time than ID3 to find a perceptron tree consistent with a set of examples. Therefore, by ranking representations based on a small number of samples ACR provides larger savings for the PT1 algorithm. This issue is discussed in more detail later.

The Tic-Tac-Toe problems

This is a set of six problems. In each problem the task is to learn to determine whether a legal, non-terminal, Tic-Tac-Toe board belongs to the corresponding category.

1. `tic1`: The set of legal boards in which the best move results in an immediate win.

2. `tic2`: The set of legal boards in which the best move is to prevent an immediate win by the opponent.

3. `tic3`: The set of legal boards in which the best move is to set up a guaranteed win in the next move (fork).

4. `tic4`: The set of legal boards in which the best move is to prevent the opponent from setting up a guaranteed win in the next move.

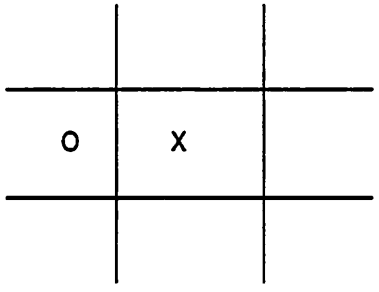


Figure 4.3 An example of a legal Tic-Tac-Toe board.

5. `tic5`: The set of legal boards in which the best move is to set up a guaranteed win in at most two more moves.

6. `tic6`: The set of legal boards in which the best move is to prevent the opponent from setting up a guaranteed win in at most two more moves.

To generate the examples for `tic`i, all the legal, non-terminal, boards that are `tic1` to `tic`$(i-1)$ are removed. The rest of the boards are classified based on whether they satisfy `tic`i. For example, to generate examples for `tic3`, boards that are `tic1` or `tic2` are removed from the set of all legal boards. The remaining boards are classified based on whether the player on move can set up a fork.

ACR was used to rank the following representations of Tic-Tac-Toe boards:

Location contents (Locs): In this representation, one numbers the squares of the board left to right, top to bottom and represents a board by giving the contents (X, O, or blank) of each square. For example, in this representation the board shown in Figure 4.3 is represented as:

$$\text{at(0, b)} \wedge \text{at(1, b)} \wedge \text{at(2, b)} \wedge \text{at(3, O)} \wedge \text{at(4, X)} \wedge \text{at(5, b)} \wedge \text{at(6, b)} \wedge \text{at(7, b)} \wedge \text{at(8, b)}$$

where $\wedge$ is the logical AND operator and b denotes a blank. This board is an example of the concept `tic5`.

Move sequence (Movs): In this representation, one numbers the moves 1 to 9 and gives the location of each move. Assuming that the first move was by X, the board given in Figure 4.3 is represented as:

$$\text{at(4, m1)} \wedge \text{at(3, m2)} \wedge \text{at(?, m3)} \wedge \text{at(?, m4)} \wedge \text{at(?, m5)} \wedge \text{at(?, m6)} \wedge \text{at(?, m7)} \wedge \text{at(?, m8)} \wedge \text{at(?, m9)}$$

where, mi denotes the i^{th} move, and a '?' denotes that the corresponding move has not yet been made.

Table 4.5 Ranking the instance representations of the Tic-Tac-Toe tasks for ID3.

Problem	ACR's prediction	ACR's savings (CPU secs.)
tic6	Movs better than Locs	$\approx$ 4 times
tic5	Movs better than Locs	$\approx$ 46 times
tic4	Movs better than Locs	$\approx$ 15 times
tic3	Movs better than Locs	$\approx$ 28 times
tic2	Neither representation better than the other	$\approx$ 9 times
tic1	Locs better than Movs	$\approx$ 18 times

Table 4.5 gives ACR's ranking of the above representations for ID3. ACR predicts that neither of the two representations is better than the other for all the six tasks taken together. The Movs representation is predicted to be better for `tic6` through to `tic3`, the Locs representation is predicted to be better for the `tic1` concept. For the `tic2` concept, after 25 runs no representation is considered to be significantly better than the other. For the `tic6` and the `tic5` tasks, the hold-out method did not produce reliable ranking of the representations. Figure 4.4 shows the accuracy of the decision trees produced by ID3 for the other four tasks.

Except for the `tic2` task, ACR's prediction agrees with the representation considered to be better based on estimating the error-rates. In addition, it is 4 to 46 times faster to rank representations with ACR than by the hold-out method. ACR's prediction that neither of the two representations is better for all the tasks taken together was verified when the accuracies were estimated. For `tic4` and `tic3` tasks, the Movs representation produced the hypothesis with the highest accuracy, while for the `tic1` task the Locs representation produced the hypothesis with the highest accuracy. For the `tic2` task, ACR did not consider either representation to be better than the other. For this task, accuracy estimation by the hold-out method showed that both the representations produce hypotheses with poor predictive accuracy, though the Locs representation produced hypotheses with a slightly higher accuracy. The reason for ACR's inability to rank representations of the `tic2` task is that with both the representations ID3 produces hypotheses with poor predictive accuracy for this task. A closer look at the theoretical results that establish the duality between data-compression and generalization show that the duality may not hold for hypotheses with very poor predictive accuracy. This issue is discussed in detail later.

Table 4.6 gives ACR's ranking of the representations of the above tasks for PT1. Again, ACR predicts that neither of the two representations is better than the other for all the six Tic-Tac-Toe tasks taken together. For the `tic2` task it predicts that the Locs representation is better for PT1, while for ID3 it had predicted that neither of the two representations was better than the other. Similarly, while for ID3 ACR predicted that the Movs representation is better for the `tic6` task, for PT1 it does not consider any representation to be better than the other. Figure 4.5 shows the

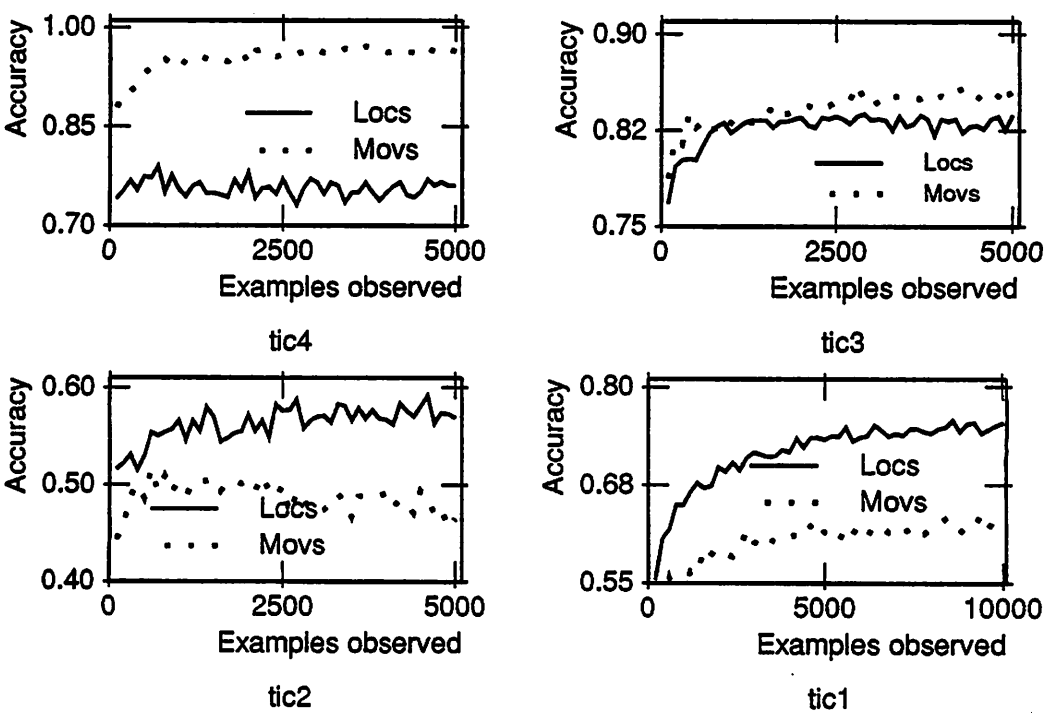


Figure 4.4 The accuracy of ID3 on test sets for the Tic-Tac-Toe tasks.

Table 4.6 Ranking the instance representations of the Tic-Tac-Toe tasks for PT1.

Problem	ACR's prediction	ACR's savings (CPU secs.)
tic6	Neither representation better than the other	$\approx$ 7 times
tic5	Movs better than Locs	$\approx$ 30 times
tic4	Movs better than Locs	$\approx$ 46 times
tic3	Movs better than Locs	$\approx$ 44 times
tic2	Locs better than Movs	$\approx$ 567 times
tic1	Locs better than Movs	$\approx$ 61 times

error-rates of the perceptron trees produced by PT1 for all Tic-Tac-Toe tasks except the `tic6` task. For the `tic6` task, we were not able to obtain a reliable estimate of the error-rate using the hold-out method. For the `tic2` task, estimating the accuracy with a single hold-out set was very time consuming, therefore cross-validation was not performed for learning this task with PT1. Even though the largest savings were obtained for this problem, because of the small difference in the accuracies with the two representations, these rankings may change with different hold-out sets. Therefore, since we are not confident that ACR's ranking for this problem will agree with a ranking obtained by using a more expensive method of estimating accuracies, from this problem we do not draw any conclusions about savings obtained by using ACR.

From this experiment we conclude that ACR's ranking of the representations agrees with the ranking produced by directly estimating the accuracies. In addition, it is faster to rank representations with ACR than by using the hold-out method to estimate accuracies. Notice that as with the Clumps tasks, for the Tic-Tac-Toe tasks also the savings obtained for PT1 are higher than the savings obtained for ID3.

Online recognition of handwritten numerals

In this class of problems, the different representations of handwritten characters were compared for learning to recognize on-line, user-independent, isolated handwritten characters. To get a set of examples for this task, seven different people were asked to write the numerals 0-9 using a mouse. Each person was asked to write every numeral several times. This process produced a set of 306 handwritten numerals. ACR was used to rank the following representations for handwritten characters.

The pixel representation: This representation was used by Casey & Nagy (1984) to describe handwritten characters to an algorithm for constructing decision trees. We describe the characters by embedding them in a 15 $\times$ 15 grid, setting each of the 225 pixels through which the pen passes to a 1 and other pixels to a 0.

The cross representation: This is a modified version of the representation used in the Ledeen recognizer (Newman & Sproull, 1979). The main idea of this representation is to divide the handwritten character into a number of strokes and extract features by detecting the regions through which the stroke passes. The regions are determined by embedding each stroke in a square. The square is divided into nine regions of equal size, as shown in Figure 4.6. To represent each stroke, one first determines the starting region and the number of times the stroke crosses each dividing line. Each stroke is represented by a 16 bit word as shown in Figure 4.7. Vertical, horizontal and very small strokes are encoded with the special codes listed in Table 4.7. Our modification was to consider each character as a single stroke and detect the regions through which the character passes. The reason for making this modification was to illustrate one possible application of ACR. One may want to determine whether making a change to some well-known representation for a task enables one to improve the accuracy of the classifiers.

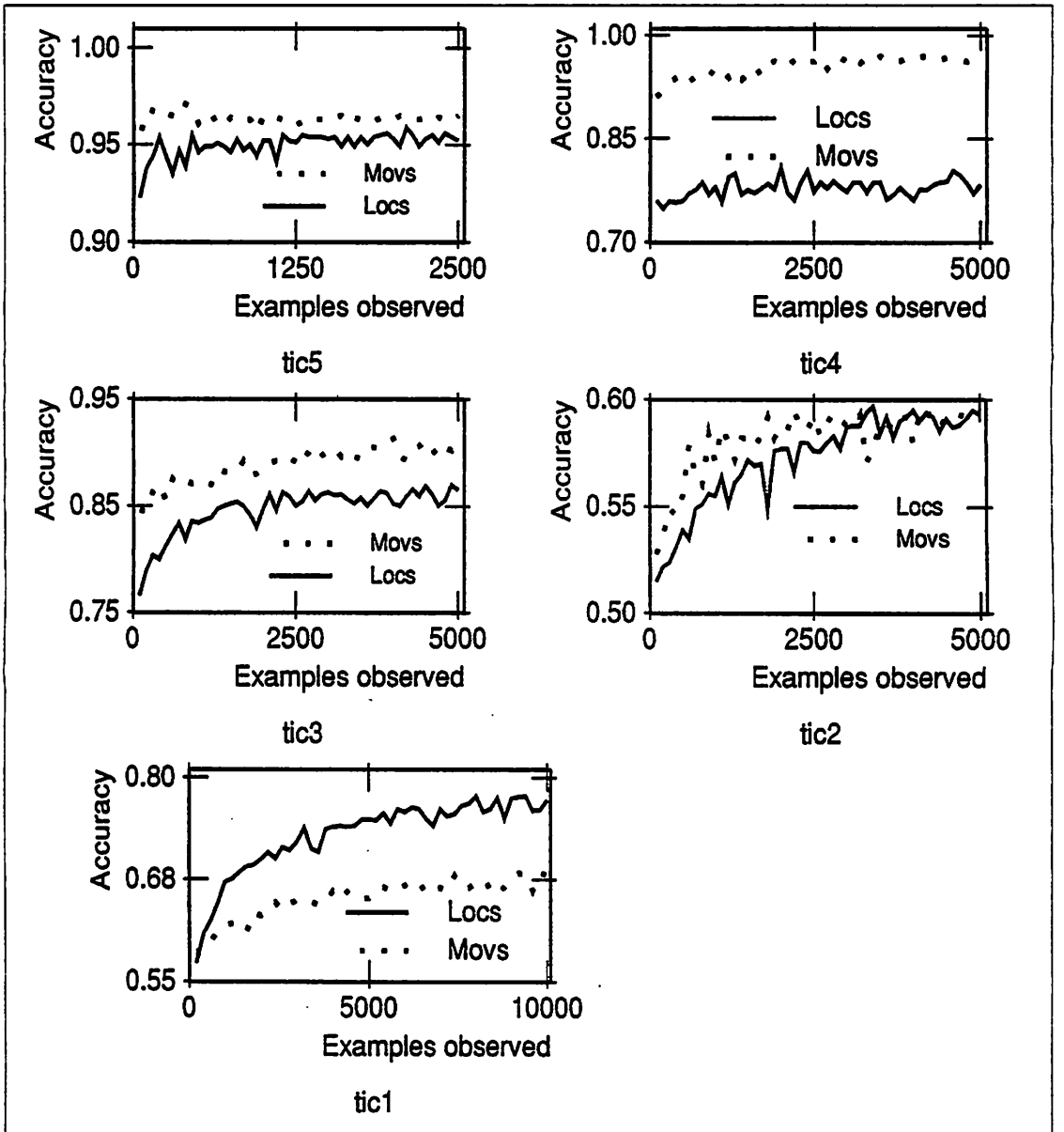


Figure 4.5 The accuracy of PT1 on test sets for the Tic-Tac-Toe tasks.

Table 4.7 Special codes for the Ledeen recognizer

Stroke	Code
Horizontal, drawn left to right	9-9-0-8
Horizontal, drawn right to left	1-1-0-8
Vertical, drawn downward	0-8-9-9
Vertical, drawn upward	0-8-9-9
Dot	0-8-0-8

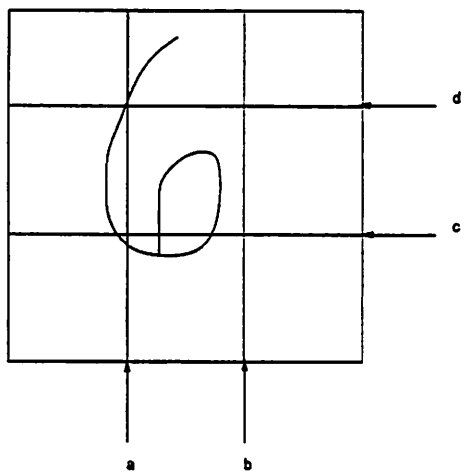


Figure 4.6 Division of the bounding square for the Cross representation.

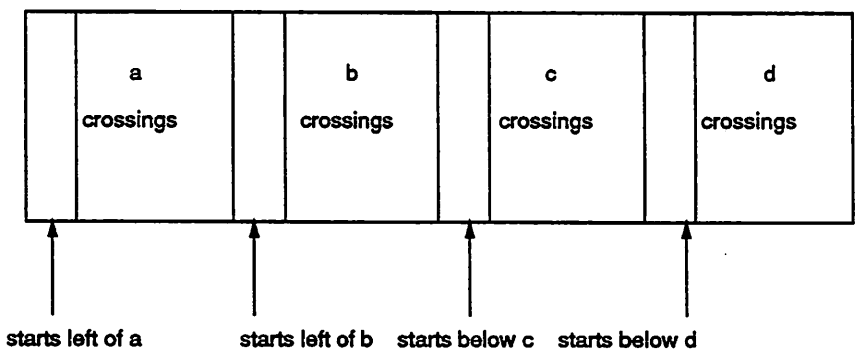


Figure 4.7 Representation of a stroke in the Cross representation.

Table 4.8 Ranking of the representations of handwritten characters for ID3

Problem name	ACR's prediction	ACR's savings (CPU secs.)
Pixel vs. Cross	Pixel better than Cross	$\approx$ 28 times
Pixel vs. Fourier	Fourier better than Pixel	$\approx$ 4 times

The Fourier representation: This representation is similar to the one used by Arakawa, Odaka & Masuda (1978) to describe the examples to a Bayesian learning algorithm. Here, one tracks the coordinates of the pen as the character is written. This gives rise to two sequences, $(X_1, X_2, \ldots, X_n)$ for the X coordinates of the pen, and $(Y_1, Y_2, \ldots, Y_n)$ for the Y coordinates. A discrete Fourier transform of these sequences is computed and the first few Fourier coefficients are used to represent the characters. In our experiments, the first 5 Fourier coefficients were used to represent a character.

The Cross and the Fourier representations contain integer and real valued attributes respectively. ID3 requires that the attributes be discrete, therefore the integer and real valued attributes are quantized by using Vapnik's method for finding the best evenly spaced discretization for an integer or a real valued attribute.

Table 4.8 gives ACR's ranking of the above representations for ID3. ACR predicts that the Pixel representation is better than the Cross representation and that the Fourier representation is better than Pixel representation. Since the "better than" relationship for representations is transitive, ACR predicts that the most accurate hypothesis from the examples will be produced from the Fourier representation. This illustrates how the current implementation of ACR can be used to rank more than two representations. Two representations can be compared at a time and the best in a set can be found by comparing the best in one comparison with one more representation.

Figure 4.8 shows the accuracy of the decision trees produced by ID3 with these representations. From this experiment we conclude that ACR's ranking of the representations for handwritten characters agrees with the ranking produced by estimating the error-rates using the hold-out method. In addition, it was faster to rank representations with ACR than by using the hold-out method. The accuracy of the decision trees produced by ID3 with the Pixel representation is not identical in the two comparisons because in each comparison the Pixel representation is compared with one other representation, and the samples presented to ID3 differ in the different comparisons. The representations for handwriting were not compared for PT1, because PT1 is designed for two-class problems and the handwriting task is a ten-class problem.

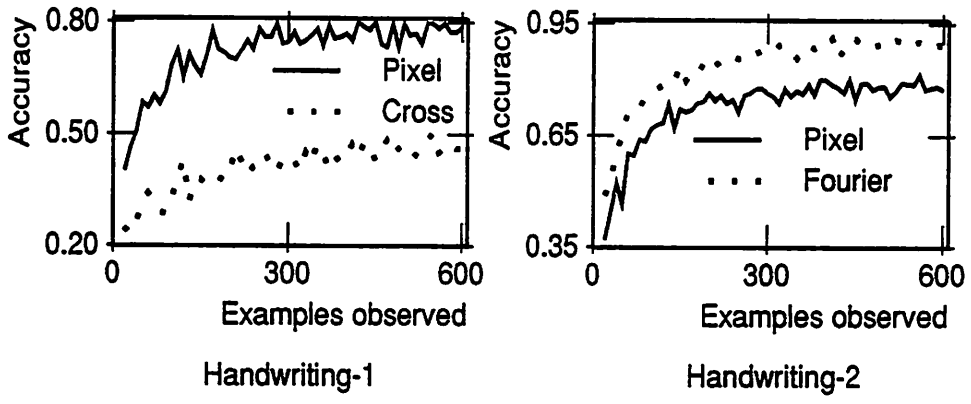


Figure 4.8 The accuracy of ID3 on the test sets.

4.1.4 Ranking Representations by Directly Estimating the Performance of the Learning Algorithm

An alternative to ranking representations by using ACR is to rank representations by directly estimating the performance of the learning algorithm with the different representations. Ranking representations by directly estimating the performance requires determining the error-rate of the hypotheses reliably. However, reliable estimation of the error-rate is expensive if a large number of examples are not available (Weiss & Kulikowski, 1991).

Figure 4.9 highlights the difficulty in reliably estimating the error-rate of a hypothesis. This figure shows the estimated accuracy of the two Tic-Tac-Toe representations for the `tic6` task. This figure shows the results of estimating the accuracy of the decision trees produced by ID3 when ten different runs were conducted with ten different hold-out sets of roughly equal size. In each hold-out set the proportion of the number of positive and negative examples present in the original set of examples was maintained. This was achieved by randomly dividing the positive examples and the negative examples separately into ten sets of roughly equal size, with each example having the equal probability of going to any of the sets. A hold-out set was produced by combining one set of positive examples with one set of negative examples. Each hold-out set was used in turn as the test set and the remaining examples were used as the training set. The accuracy estimation was done as in Section 4.1. The first and the tenth hold-out set indicate that the Locs representation is better than the Movs representation. For the fourth hold-out set the best estimated accuracy is the same for both the representations, though usually the Locs representation produces hypotheses of higher estimated accuracy. From all the other hold-out sets, one concludes that the Movs representation is better than the Locs representation. For this problem ACR predicts that the Movs representation is better based on all 1026 examples and also based on each of the ten training sets corresponding to the ten hold-out sets.

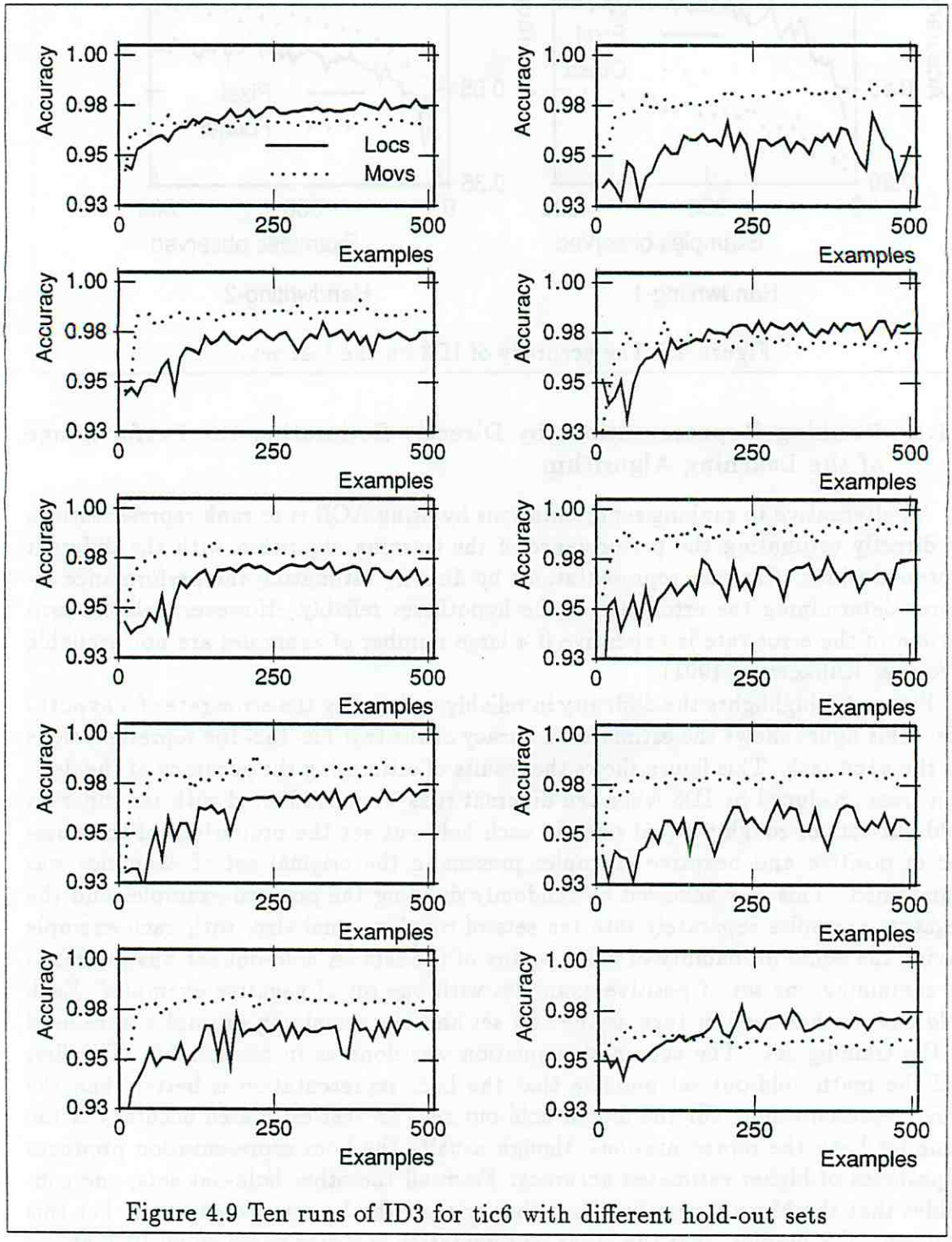


Figure 4.9 Ten runs of ID3 for tic6 with different hold-out sets

The above example illustrates how inexpensive techniques for error-rate estimation, like the hold-out method, can lead to unreliable results when one has a limited number of examples. In these situations, one has to resort to expensive re-sampling methods to arrive at accurate error-rate estimates. The above procedure, of using ten different hold-out sets is an example of these expensive re-sampling methods. It is one way of using ten-fold cross-validation to rank representations.

The previous section showed the savings obtained by using ACR rather than estimating error-rates of hypotheses with a single hold-out set. Other methods for estimating the error-rate, like the leave-out-one and k-fold cross-validation are more expensive than the hold-out method. Consequently, one can expect larger savings over these methods when ACR is used to rank representations.

Another example that illustrates that fast methods for estimating error-rates can result in inaccurate results is ranking representations of the `tic6` task for PT1. Figure 4.10 shows the results of estimating the accuracies of perceptron trees produced by PT1 with the same test sets as above. For this problem ACR predicts that neither the Movs nor the Locs representation is better than the other. Estimating the error-rates shows that accuracy of the hypotheses produced with the different representation is indeed very close. For three of the ten cases, the two representations produce almost identical accuracies. In one case the Locs representation is better. In the other six cases, the Movs representation is slightly better. Taken together, from this experiment we cannot make any definitive conclusion about which representation is better for PT1.

4.2 Analysis of the Results

The above experiments provide empirical support for concluding that ACR is effective in solving the representation comparison problem for a variety of learning tasks and learning algorithms. These results show that even in situations where relatively inexpensive methods for estimating error-rate— like the hold-out method— provide accurate estimates of the error-rate of a hypothesis, ranking representations by using ACR is faster. In the situations where the hold-out method did not provide accurate estimates of the error-rate, the ranking produced with ACR agreed with the ranking obtained by using more expensive methods for estimating the error-rate. In these situations, the savings obtained by using ACR rather than directly estimating accuracies was considerably more.

In our experiments it was observed that the savings obtained by using ACR are much more for PT1 than they are for ID3. This is because the time required by PT1 to produce a consistent hypothesis is usually much more than the time required by ID3. Since ACR can often pick the better representation by observing a small number of samples, this translates to a larger savings for PT1. This suggests that the savings obtained by using ACR to rank representations will increase as the time

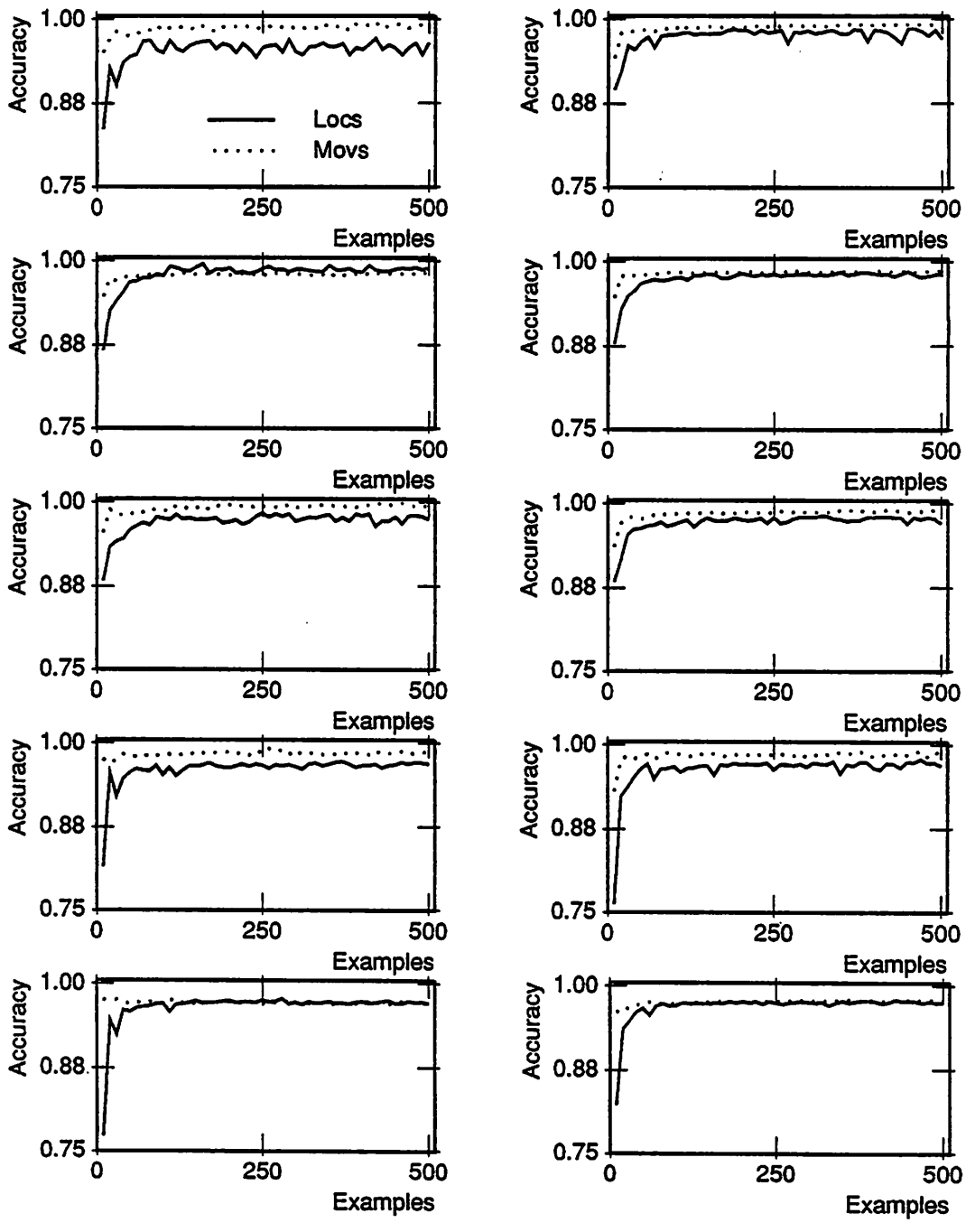


Figure 4.10 Ten runs of PT1 for tic6 with different hold-out sets

complexity of the learning algorithm increases. In situations where it is necessary to have high accuracy in classifying the instances, one may choose to use a learning algorithm with a large time complexity. In these situations, using ACR to identify the better representation will result in increased savings.

The reason that ACR correctly ranks instance representations is its use of compact coding as a basis for ranking representations. Our experiments show that the correspondence between data-compression and generalization is precise enough that even small differences in accuracy can be detected by determining the minimum codelength for the examples. Also, since the minimum codelength is estimated from all the available examples one does not face the problem of producing a representative testing set. The problems associated with producing such a testing set are the reason why fast methods for estimating error-rate are inaccurate and accurate methods are time-consuming.

The reason for ACR's inability to rank representations of the `tic2` task for ID3 is explained by a closer look at the theoretical results that establish the duality between data-compression and generalization. In the `tic2` task one is trying to learn the function that tells whether the best move available is to prevent an immediate win by the opponent. Since the task is to learn a function and not to model a probabilistic phenomenon, the minimum description length principle cannot be applied. Recall from Section 2.2 that algorithmic information theory establishes the duality only for hypotheses that do not have very poor predictive accuracy. In the `tic2` task the decision trees that result from both the representations have a poor predictive accuracy. Therefore, duality between data-compression and generalization established by algorithmic information theory does not hold in this case. The poor predictive accuracy of the hypotheses indicates that the space of hypotheses considered by ID3 is large. For this large space of hypotheses one cannot have high confidence in the accuracies of the hypotheses without observing a large number of examples. Therefore, given the fixed number of examples and a large hypothesis space, one can only assert with a low confidence that a compact hypothesis will have a higher accuracy.

There are two reasons why ACR is faster than methods for ranking representations by estimating error-rates. The first reason is the effectiveness of the two tests for terminating a run. With these tests, ACR can terminate a run by observing only a few samples from the examples. The number of samples required depends on whether the representations being compared are well-suited for the task. Bad representations are identified as such with a small number of samples. If both representations being compared are well-suited to the task, then the number of samples depends on the amount of difference in the accuracies produced with the different representations. The second reason why ACR is faster is that the ranking obtained in a run usually does not change in multiple runs. As a result, only a few runs are required to obtain the desired confidence in the ranking. That is, codelength was found to be a measure of predictive accuracy that is robust against sampling errors.

4.3 Other Approaches for Ranking Representations

Very few techniques for comparing representations are known. ACR is closest in spirit to an experiment reported by Gao & Li (1989). They used the data compression view to design a classifier for handwritten alpha-numerals. Their approach was to construct a database of prototypes for each character from a training set that consisted of handwritten input along with the character it represents. To classify an input, its distance from the prototypes of each character was measured, and the input was classified to be the character that had a prototype closest to the input. To construct the prototypes and to classify the inputs, pen coordinates were recorded at regular intervals. The recording interval was called the *feature extraction interval.* The problem was to determine the best feature extraction interval. Their approach was to try a number of different intervals. For each interval, they computed the sum of the number of bits required to store the prototypes and the number of bits required to store the characters in the training set that are mis-classified by their classification procedure. They showed that the feature extraction interval for which the above sum was minimized produced prototypes with the highest predictive accuracy over an independent test set.

In presenting their work Gao and Li say that they use the description length criterion to select among models in a family of models where each member of the family is generated by a particular feature extraction interval. However, if one views the choice of feature selection interval as a choice of instance representation, then determining the best feature selection interval can be viewed as selecting the best representation.

Here, we have taken this view and generalized it substantially so that we now have an algorithm to identify the best representation for any learning algorithm that satisfies the assumptions for the two tests and for which we have a method for determining the codelength for the hypotheses. Also, unlike Gao and Li's approach, ACR attempts to determine the minimum codelength required to describe the examples, rather than find the codelength of a hypothesis constructed in one particular manner, namely from all the available examples. In terms of our approach, Gao and Li's approach is equivalent to deciding the better representation based on a single sample. On the other hand ACR ranks representations based on the best accuracy that can be achieved with a given representation. The two tests for terminating a run enable one to estimate this accurately without observing samples of all sizes, which leads to savings in computational time required to rank representations.

In our earlier attempts (Saxena, 1989; Saxena, 1990) we ranked the representations for learning discrete functions based on the size of the minimal circuit required to express the examples in the various representations. From the data-compression viewpoint, this approach is justified because the size of the minimal circuit required to implement a discrete function is a good approximation to the algorithmic information of the function (Abu-Mostafa, 1986). Unfortunately, it is intractable to find

the size of the minimal circuit required to implement a function, so we approximated the size of the minimal circuit by the size of the minimal *disjunctive normal form* (DNF) expression. However, even finding the minimal DNF expression for a function is NP-complete, so we had to approximate the minimal DNF expression with a small DNF expression.

While this approach produced empirically good results, it was unsatisfying for the following reasons. First, by ranking the representations based on circuit size, and therefore on algorithmic information, one determines which representation is better when any program for a universal Turing machine can be a hypothesis. But, in practice this is never the case. To keep matters computable and tractable, a learning algorithm uses a restricted model class and searches it in a specific manner. The problem of determining the better representation for a specific learning algorithm is therefore more relevant. To do this one needs to determine the efficiency with which a particular algorithm of interest compresses the observations. Second, by constructing a minimal DNF expression one introduces a bias into the determination of the circuit size. There are functions with small circuits that have large DNF expressions, for example, the parity function. That is, one cannot guarantee that if one function has a smaller DNF expression than another, it will necessarily have a smaller circuit.

The problem of evaluating individual features has received more attention. In machine learning, the earliest work on feature evaluation appears to be in Samuel's (1959) checkers playing program. A set of 38 board features were available to a program that learned an evaluation function for the boards. Only 16 of these features were used for learning an evaluation function. The utility of a feature was determined by observing whether there was any correlation between the value of the feature and the evaluation of the board. The features with the highest utility were used for constructing an evaluation function. It is not clear how this approach can be utilized to rank representations, because the correlation of a set of features with the value of an example may not be well-defined, especially when the attributes are discrete and the task is to learn to classify the instances into a number of categories.

An extensive body of work in statistical pattern recognition tries to evaluate a feature based on how well it discriminates the space of possible instances. This is useful for finding a small set of features that allow a low probability of mis-classification. Kittler (1986) gives a survey of these approaches. The goal of these techniques is to find a set of features with which a classifier with the least possible error rate can be constructed, given knowledge of the probability density functions required to construct the optimal classifier. Examples of these techniques are methods that employ *probabilistic distance measures*, for estimating the "distance" between the probability densities of observing different classes conditioned on an observation; *probabilistic dependence measures* for measuring the degree of dependence between a class and an observation; and *entropy measures* for measuring the information gained about the classification from an observation. However, the goal of representation evaluation is to determine which representation enables one to produce a classifier with the least

error with the *given* examples and for the *given* learning algorithm, rather than a classifier with the lowest possible error. Therefore, feature selection methods are not suitable for representation comparison for the following reasons. Firstly, when learning deterministic concepts the examples can be perfectly discriminated with the best possible classifier in each representation, and therefore according to the above criteria for evaluating features all representations are equally good. Secondly, usually one does not have access to the large number of examples required for determining the various probability density functions. Finally, density estimation is computationally expensive.

A number of heuristic inter-class distance measures have also been proposed for feature selection. However, in general their relationship to error probability is not always clear. Mehra et al. (Mehra, Rendell & Wah, 1989) have proposed a heuristic measure that considers a feature to be good if it has the same value for examples belonging to the same class and different values for examples belonging to different classes. However, it is not clear how one can combine the two requirements into a single measure of feature-worth. Also, it is not clear how this measure should be generalized to rank representations rather than single features. This measure also depends on a notion of distance between different values of a feature and for categorical variables it may be difficult to decide the appropriate distance metric.

Rendell and Seshu (1990) propose using a heuristic measure of concept complexity, called *concept variation* to rank representations. The variation of a boolean function is the sum of the number of hamming-one neighbors of every example that have different function values. As the authors recognize, this measure has a number of deficiencies. Specifically, this measure requires multi-valued features to be converted to binary values with all possible orderings for the attributes. If there are a large number of attributes and each attribute can take a large number of values, this procedure is very expensive because one would need to try all possible orderings of all the variables. Also, when only a small fraction of the instances are observed, most of the hamming-one neighbors are unobserved and in that case it is not clear what values should be assigned to the unobserved positions. If the unobserved positions are ignored then concept dispersion is not a reliable measure of inductive difficulty.

4.4 Summary of the Chapter

This chapter presented the results of using ACR to solve eighteen instances of the representation comparison problem. These eighteen instances involved the ID3 and the PT1 learning algorithm and three classes of learning tasks. These tasks were the Clumps problems, the Tic-Tac-Toe problems, and the Handwriting recognition problems. In sixteen of the eighteen cases, ACR identified the representation that produced the most accurate hypothesis. This was confirmed by directly estimating the accuracy of the hypotheses produced by the learning algorithm. In two cases,

ACR did not consider either representation to be better than the other. In one of these two cases, namely `tic6` for PT1, based on ten-fold cross-validation it could not be concluded that one representation is definitely better than the other. In the other case, namely `tic2` for ID3, estimating the accuracy of the hypotheses produced by the learning algorithm showed that for both the representations ID3 produced decision trees with poor predictive accuracies. A closer look at the theoretical results that establish the duality between data-compression and generalization show that the duality does not hold for hypotheses with very poor predictive accuracy. Taken together, the above results support the conclusion that ACR is effective in ranking representations as long as not all the representations being compared are poorly suited for the task.

In addition, it was 4 to 70 times faster to rank representations by using ACR than by using the hold-out method to estimate the accuracy of the hypotheses produced with the different representations. Savings of 4 times were obtained when both the representations being compared were well-suited for the task but one representation was slightly better than the other. Savings of 70 times were obtained when one representation was clearly better than another. Since the hold-out method is a quick (though potentially inaccurate) method of estimating accuracies, one can expect the saving to increase if more expensive, albeit accurate, methods of estimating error-rate are used. This chapter also illustrated how inexpensive methods for error-rate estimation can result in wrong conclusions about the quality of a representation, making the use of expensive methods necessary. ACR's effcetiveness in solving the eighteen problems reported in this chapter, which represented a variety of learning situations, and the theoretical arguments establishing the duality between data-compression and generalization, suggest that ACR will be effective in solving the representation comparison problem for many other learning tasks. In addition, one can expect savings in ranking representations by using ACR than by directly restimating the performance of the learning algorithm with the different representations.

CHAPTER 5

TWO TYPES OF REPRESENTATION CHANGE

The previous chapter presented a number of tasks and multiple instance representations for each task. In most cases, one of the representations was better for inductive learning than the other representations for that task. The conclusion from ACR's effectiveness in ranking representations was that the best representation enables the learning algorithm to describe the classification of the examples most compactly. This chapter addresses the issue of whether in each of the cases where one representation was better than the other, the reason for the increased compressibility was the same. To address this question we observed the effect of representation change on two vastly different types of learning algorithms, namely ID3 and PT1 on one side and *rote learning* on the other. In rote learning, one remembers the observed examples. To determine the value of the function being learned for a specified input, one first determines if that value was observed before. If it was, then the observed output is returned, otherwise a special value called "unknown" is returned. Changing the representation will affect the accuracy of rote learning if one representation enables the examples to be remembered with a fewer number of bits than the other.

From all the examples of representation change given in the last chapter the two-clumps and the `tic4` task were selected. In both these tasks changing the representation considerably effects the accuracy of the hypotheses produced by the learning algorithm. In the two-clumps task, the Edge representation was found to be much better than the Pixel representation for both ID3 and PT1, and in the `tic4` task, the Movs representation was found to be much better than the Locs representation for both ID3 and PT1. If the reason for the improved compressibility is the same in both these tasks, then we expect that if the Edge representation is better than the Pixel representation for rote learning the two-clumps task, then it must also be the case that the Movs representation is better than the Locs representation for rote learning the `tic4` task. Section 5.1 shows that this is not the case; it then goes on to explain this difference. The explanation is based on the observation that different representation changes improve compressibility due to different reasons. This suggests that representation changes can be classified on the basis of the mechanism by which they improve the compressibility of the examples. Section 5.2 takes a step in this direction. It presents two methods for producing a good representation based on two different techniques for improving the compressibility of the examples.

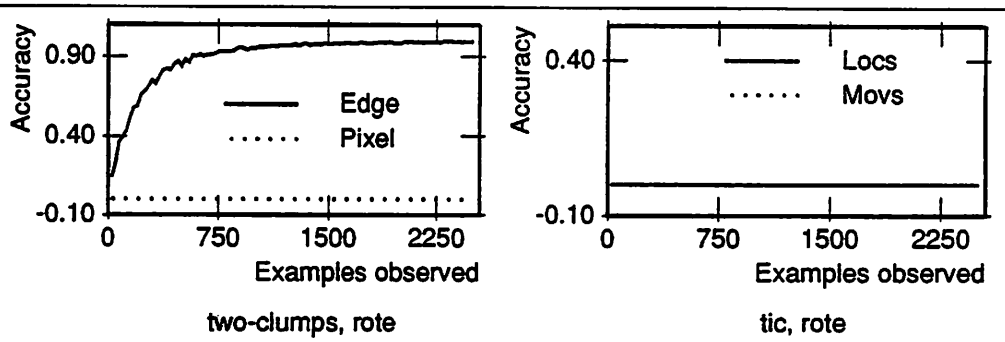


Figure 5.1 The effect of the change of representation on rote learning.

5.1 Randomness of a Function with Respect to a Hypothesis Class

Identical change of representation affects different learning algorithms differently. Figure 5.1 shows the effect of changing the representation of the Tic-Tac-Toe and the Clumps problem on *rote-learning*. This graph shows that for the Clumps task the Edge representation is better than the Pixel representation for rote learning. There is no generalization with the Pixel representation because there are no examples common to the training and the testing sets. Rote learning generalizes with the Edge representation because a number of examples in the Pixel representation map to the same example in the Edge representation. As a result there are examples common to the training and testing sets in the Edge representation. However for the Tic-Tac-Toe task, there is no difference in the generalization achieved with the Movs and the Locs representation. For both representations of the Tic-Tac-Toe task, the accuracy of the hypotheses produced is always zero percent.

Figure 5.1 and the results of the previous chapter show that for the Clumps task the Edge representation is better than the Pixel representation for rote learning, ID3, and PT1. However, while the Movs representation of the tic4 task is better than the Locs representation for ID3 and PT1, that is not the case for rote learning. This figure also shows that the same change of representation for the Tic-Tac-Toe task has a considerable effect on ID3 and PT1 and has *no* effect on rote learning. On the other hand, changing the representation of the Clumps task affects rote learning *more* than it affects ID3 and PT.

The reason why the change of representation of the two-clumps problem affects rote learning and the change of representation of the `tic4` task does not, is that these are two distinct types of representation change. The difference in these two types of representation change can be explained with the help of Abu-Mostafa's (1986,1988) notion of *deterministic entropy* of a boolean function. Abu-Mostafa defines a number of complexity measures for boolean functions, of these the following are of special interest:

1. The *deterministic entropy* of a boolean function f, denoted by $H(f)$.

2. The *randomness* of a boolean function f, denoted by $R(f)$.

The definition of these measures given here is slightly different from Abu-Mostafa's definitions. The value of his measures is the logarithm of the values defined below. For purposes of this discussion, taking the logarithm does not make any difference.

If the output column of the truth table [1] of a boolean f has $h_1(f)$ 1's and $h_0(f)$ 0's then the *deterministic entropy* of f is defined to be,

$$H(f) = \min(h_0(f), h_1(f)).$$

The *randomness*, $R(f)$, of a boolean function is the algorithmic information of the string formed from the output column of the truth table of the function.

Abu-Mostafa's complexity measures are defined for an arbitrary but fixed number of inputs N. However, functions that result from different representations may differ in the number of inputs. In order to compare the complexity of two functions with a different number of inputs, we make the number of variables in the two functions equal by transforming the function with the fewer input variables into an equivalent function with additional dummy variables. If k dummy variables are added, then each input pattern of the original function produces 2^k input patterns in the function that results from the transformation. Of these 2^k patterns, the first one in lexicographic order takes the value of the original function and the rest take the majority value in the original function. Notice that this does not change the value of $H(f)$ at all. The value of $R(f)$ increases by at most the constant amount that it takes to specify the sentence, "first generate the output column and then pad it as above."

$H(f)$ and $R(f)$ give upper and lower bounds respectively on the amount of information needed to compute f. A function is known completely if all the inputs for which it takes the value 1 (or 0) are known. For a function of an arbitrary but fixed number of inputs N, this can be done with $O(H(f))$ bits. Similarly, by the definition of algorithmic information, f cannot be computed by any effective procedure that can be specified with fewer than $R(f)$ bits.

If one views learning a function f as a process of extracting from the examples the information required to compute f, then $H(f)$ is the least number of bits that must be extracted from the examples if only rote learning is permitted. Likewise, $R(f)$ is the least number of bits that must be extracted if one is given the full power of a universal Turing machine for the purposes of constructing hypotheses. The important thing to note is that $H(f)$ is an upper bound on $R(f)$.

We generalize the notion of randomness of a function to include the particular hypothesis class M employed by a learning algorithm. For instance, M can be the class of all decision trees. Denote by $R_M(f)$ the size (in bits) of the most compact hypothesis in the class M that expresses the function f. This notion is similar to

[1] A truth table is an enumeration of the value of the function for all the possible inputs, where the enumeration is lexicographic based on the values of the inputs.

Rissanen's (1989) notion of minimum description length of a set of observations with respect to a class of probabilistic models.

For a learning algorithm that produces the most compact hypothesis in the class M consistent with any set of examples, $R_M(f)$ gives the minimum amount of information the learning algorithm must extract from the examples in order to compute f. A good instance representation reduces the amount of information required to learn f by reducing the value of $R_M(f)$.

5.2 Two Types of Representation Change

The randomness of a function can be reduced in at least two ways. The first is by reducing deterministic entropy. Suppose that f_1 and f_2 are the functions that result from the representations R_1 and R_2. If $H(f_2) < R(f_1)$, then $R(f_2) < R(f_1)$, because $H(f)$ is an upper bound on $R(f)$. We call this type of representation improvement *dimensionality reduction.* It is achieved by collapsing a number of inputs of the function that results from the worse representation into a single input of the function that results from the better representation.

The Clumps problem exemplifies dimensionality reduction. The deterministic entropy with the Pixel representation is 495 bits and with the Edge representation is 45 bits. By going from the Pixel to the Edge representation, one collapses a number of examples in the Pixel representation to a single example in the Edge representation. This is the reason why the Edge representation is better than the Pixel representation for rote learning. The amount of information that needs to be extracted is less in the Edge representation than in the Pixel representation. By acquiring the ability to correctly classify one example in the Edge representation, one acquires the ability to classify all the corresponding examples in the Pixel representation. The reduction in the deterministic entropy that accompanies the change from Pixel to Edge representation is large enough to imply a decrease in the randomness of the corresponding functions for decision trees and perceptron trees. This reduction in randomness explains why Edge representation is better than Pixel representation for ID3 and PT1.

Dimensionality reduction is not the only way to improve a representation. In the Tic-Tac-Toe task, for both the Movs and the Locs representations the deterministic entropy of the resulting function is the same. It is 206 bits for both the representations. This is the reason why changing the representation from Locs to Movs does not affect rote learning. The reason that Movs is better than Locs for ID3 and PT1 is that the value of $R_M(f)$ for the function that results from the Movs representation is lower than the value of $R_M(f)$ for the function that results from Locs representation.

Unlike dimensionality reduction, we do not know the exact mechanism by which the randomness is reduced in the Tic-Tac-Toe task. Therefore, we will continue to call this second type of representation change *randomness reduction.* In this kind of

representation change, one does not improve a representation by collapsing a number of cases into a single case, and thereby reducing the number of possibilities to be considered. Instead, one improves the representation by exposing some regularities in the examples in some other manner that can be utilized by the learning algorithm to produce a compact hypothesis for the examples, which in turn results in accurate predictions.

5.3 The Interaction between the Hypothesis Class and the Instance Representation

Changing the representation of the Clumps problem affects the accuracy of rote learning more than the accuracy of hypotheses produced by ID3 or PT1. This is because the concept description language employed by rote learning is so improvised that it cannot express the simple computation required to translate the Pixel representation to the Edge representation. However, decision trees and perceptron trees can count the number of transitions from a black to a white Pixel and, consequently, changing the representation affects ID3 and PT1 less than it affects rote learning. By the same token, one can expect that changing of representation of the Tic-Tac-Toe task will affect ID3 and PT1 more than it would affect a learning algorithm that can produce any program for a UTM as a hypothesis. The simple program, "convert the given set of location contents into move sequences in all possible ways, and choose the shortest hypothesis produced from these sets", enables a learning algorithm that employs UTM's as a concept description language to first convert the boards in the location contents representation to the move-sequence representation and then build a compact hypothesis. That is, the value of $R(f)$ for the function that results from the two representations is approximately equal.

The Tic-Tac-Toe and the Clumps examples also suggest what the effect of the change of representation would be along the spectrum from trivial hypotheses produced by rote learning to all possible computations produced by UTMs. Initially, the increased expressiveness will make it possible to utilize the regularities exposed by the change of representation to find a compact hypothesis for the examples. As a result, a representation change will produce large differences in the accuracy of the hypotheses that result from different representations. However, as the expressiveness of the hypothesis class increases, it will become possible to express compactly the computation required to convert from one representation to the other. Consequently the difference in the randomness with the different representations will not be significant, resulting in little difference in the accuracy of the hypotheses produced with different representations.

5.4 Summary of the Chapter

This chapter identified two types of representation change based on two different methods of improving compressibility of the examples. The first type of representation change was called dimensionality reduction. This type of representation change occurs in the Clumps problems between the Pixel and the Edge representations. A number of inputs of the Pixel representation are mapped to a single input in the Edge representation. Dimensionality reduction results in a function with a lower deterministic entropy. Since the deterministic entropy of a function is an upper bound on the randomness of a function, a large enough decrease in the deterministic entropy results in a function with a lower randomness, or equivalently increased compressibility.

However, in all the Tic-Tac-Toe tasks the deterministic entropy of the functions that result from both the representations is the same. One representation was better than another in these tasks because the better representation reduced the randomness of the examples by exposing some regularities. These regularities were utilized by the learning algorithm to find a compact description for the examples. This type of representation change was called randomness reduction. We believe that identification of other methods of improving compressibility will result in a further classification of the types of representation change. Such a classification can provide a set of methods for designing good representations.

CHAPTER 6

CONCLUSIONS

The objectives of this research were to understand why instance representations affect generalization and to determine how one can rank instance representations on the basis of their suitability for learning from examples. Our approach for answering these questions was to test the hypothesis that representations can be ranked based on the minimum number of bits required to describe the examples in the different representations. This chapter summarizes the results obtained in this thesis and discusses how they support the hypothesis. Finally, we conclude with a discussion of some promising directions for future research.

6.1 Summary of the Dissertation

The hypothesis that representations can be ranked by measuring the minimum number of bits required to describe the examples in the different representations was suggested by the duality between data-compression and generalization. This duality states that the more compactly one can describe a set of observations with a model, the more accurate will be the predictions made with that model; and conversely, if there is no model that enables the examples to be described compactly, then one will not be able to make good predictions from the observations. The different arguments that establish this duality are summarized below.

6.1.1 Theoretical Basis

Section 2.2 reviewed the results obtained by the application of algorithmic information theory to inductive inference. These results show that if any program for a universal Turing machine (UTM) can be a hypothesis, then the probability of the hypothesis being correct increases as the size of the shortest program required to compute the hypothesis on the UTM decreases. For the same task, different instance representations result in different functions. The size of the most compact hypothesis differs when the examples are drawn from different functions. Among these hypotheses, the hypothesis that can be computed with the fewest bits is most likely to be the correct hypothesis.

Unfortunately, the size of the shortest program to compute a function on a UTM is uncomputable in general. So, one cannot use the size of the shortest program for a hypothesis as a basis for ranking representations. In addition, to make hypothesis selection tractable and computable, learning algorithms do not consider all programs for a UTM as hypotheses. They either restrict the space of possible functions or the space of possible programs. The results from learnability theory establish the duality between data-compression and generalization in a few of these situations.

Section 2.3 reviewed the results from learnability theory that establish the duality between data-compression and generalization for the class of learning algorithms that search a space of hypotheses in the strict order of increasing size of the hypotheses. Algorithms that generalize by finding a compact hypothesis consistent with the examples can be viewed as searching the space of hypotheses in the order of increasing complexity. Results from learnability theory show that in order to produce with high confidence a hypothesis with the desired predictive accuracy, it suffices to select any consistent hypothesis from a small class of hypotheses. In addition, for a fixed number of examples, as the number of concepts in a class increases, the chances increase that the learning algorithm will produce a hypothesis with poor predictive accuracy for at least one member of the class. Application of the results from learnability theory to the class of algorithms that search a space of hypotheses in the order of increasing complexity shows that one's confidence in the accuracy of the hypothesis produced by the learning algorithm increases with the compactness of the hypothesis. In addition, if these algorithms produce a complex hypothesis, then it is likely that the complex hypothesis will have poor accuracy.

The size of the most compact consistent hypothesis for the examples differs for different instance representations. If by using instance representation R the learning algorithm produced a hypothesis more compact than the hypotheses produced by using other instance representations, then the learning algorithm had to consider fewer hypotheses when R was the instance representation. Therefore, with high likelihood, the most accurate hypothesis will be produced with R and not with the other representations.

Section 2.4 reviewed minimum description length results from statistical estimation theory. These results show that for probabilistic phenomena, the model that enables the observations to be described most compactly is the maximum-likelihood model for the observations. Different instance representations result in different models for the underlying data-generating machinery. The maximum likelihood model for a set of observations results from the instance representation with which the observations can be described most compactly.

6.1.2 ACR: The Design and Implementation

The above results prompted the development of ACR, an algorithm to compare representations. However, these results were only suggestive of a criterion for comparing instance representations. Each of the results discussed above is valid only

under specific conditions. In practice it may be difficult to ascertain whether these conditions hold exactly or only approximately. For instance, practical learning algorithms do not consider all possible programs as hypotheses. They usually have a fixed concept-description language. Therefore, strictly speaking, results from algorithmic information theory do not justify our use of compactness of a hypothesis as a basis for comparing representations. Similarly, even though a number of algorithms generalize by producing a compact hypothesis for the examples, one cannot say that these algorithms search the space of hypotheses in a strict order of increasing complexity. Therefore the results from learnability theory cannot be applied directly. Also, often we compare representations for learning functions rather than for modeling probabilistic phenomena. In these situations, the various quantities required in the precise formulation of the minimum length description principle may not exist. Therefore, we consider ACR to be a further explication of the duality between data-compression and generalization, rather than being a direct application of any of the formal results mentioned above.

ACR ranks representations by estimating the minimum number of bits required to specify a consistent hypothesis for the examples in the different representations. To estimate the minimum codelength efficiently, ACR observes the trend in the codelength as an increasing number of examples are presented to the learning algorithm. After observing a sample from the examples, ACR conducts tests to determine whether the codelength is likely to increase with sample size and whether the codelength is likely to decrease further. If these tests indicate that the codelength is likely to increase with sample size, then ACR concludes that presenting more examples to the learning algorithm is unlikely to reduce the codelength. Similarly, if these tests indicate that the codelength is unlikely to decrease further then ACR stops presenting examples to the learning algorithm. Representations are ranked based on the estimated size of the minimum codelength for different representations.

Section 3.1.2 discussed a test that indicates whether the codelength is likely to increase with increasing sample size. This test is based on the fact that if the codelength did not increase with sample size, then by increasing the sample size the codelength is equally likely to increase or decrease. However, if ACR observes that in a large number of cases increasing the sample size increases the codelength, it determines the probability of this event occurring by chance and concludes that the codelength increases with sample size if this probability is small. Section 3.1.3 discussed a test that indicates whether the codelength is likely to decrease further if additional examples are presented to the learning algorithm. This test is based on the fact that if one can determine the probability of the occurrence of any particular error-list, then given the assumptions required for the test, the codelength cannot decrease further by presenting additional examples to the learning algorithm. However, since the probability of occurrence of an error-list is unknown, ACR uses an approximate test of independence to estimate the probability of occurrence of independent error-lists.

6.1.3 Results Obtained by Using ACR to Rank Representations

Chapter 4 presented the results of using ACR to solve eighteen instances of the representation comparison problem. These eighteen instances involved the ID3 and the PT1 learning algorithm and three classes of learning tasks. The first class of tasks consisted of learning to recognize patterns of contiguous blocks of black pixels in a one-dimensional visual field. The second class of tasks involved learning to recognize the best move available in a Tic-Tac-Toe board. The third class of tasks involved learning to recognize handwritten characters. ACR was validated by using it to rank representations, and then validating its prediction by directly estimating the accuracy of the hypotheses using the hold-out method. If the difference in the accuracies obtained with the different representations was small, ten-fold cross-validation was used to confirm the ranking obtained from the hold-out method. In sixteen of the eighteen cases, ACR identified the representation that produced the most accurate hypothesis. In two cases, ACR did not consider either of the two representations being compared as being better than the other. In one of these two cases, namely `tic6` for PT1, based on ten-fold cross-validation it could not be concluded that one representation is definitely better than the other. In the other case, namely `tic2` for ID3, estimating the accuracy of the hypotheses produced by the learning algorithm showed that for both the representations ID3 produced decision trees with poor predictive accuracies. A closer look at the theoretical results that establish the duality between data-compression and generalization show that the duality does not hold for hypotheses with very poor predictive accuracy. Taken together, the above results support the conclusion that ACR is effective in ranking representations as long as not all the representations being compared are poorly suited for the task.

In addition, ranking representations by using ACR was 4 to 70 times faster than ranking them by using the hold-out method of estimating accuracy of a hypothesis. Savings of 4 times were obtained when both the representations being compared were well-suited for the task but one representation was slightly better than the other. Savings of 70 times were obtained when one representation was clearly better than another. Since the hold-out method is the cheapest (though potentially inaccurate) method of estimating accuracies, one can expect the savings to increase if more expensive, albeit accurate, methods of estimating error-rate are used. The exact amount of savings depends on the particular set of representations being compared and the learning algorithm. Bad representations are identified as such with a small number of samples. If both representations being compared are well-suited to the task, then the number of samples depends on the amount of difference in the accuracies produced with the different representations. Also, because ACR rank representations by observing a small number of samples from the examples, the savings increase as the time-complexity of the learning algorithm increases. Chapter 4 also illustrated how inexpensive methods for error-rate estimation can result in wrong conclusions about the quality of a representation, making the use of expensive methods necessary. Taken together, the conclusion from our experiments was that ACR is effective in solving

the representation comparison problem, and that it is faster to rank representations by using ACR than by estimating the error-rate of the hypotheses produced with the different representations.

6.1.4 Categorizing Representation Change on the Basis of Compressibility

The theoretical results of Chapter 2 and the results obtained by using ACR to rank representations support the conclusion that one representation is better than another because the examples can be described more compactly in the better representation. This suggested that different methods for improving compressibility will give rise to different techniques for producing good representations. Chapter 5 identified two distinct methods for producing a good representation based on two different methods of improving compressibility of the examples. The first method was called dimensionality reduction. This method of producing a good representation was exemplified by the Clumps problem. The Edge and the Pixel representation of this problem differ because a number of inputs of the Pixel representation are mapped to a single input in the Edge representation. Changing a representation by reducing the dimensionality results in a function with a lower deterministic entropy. Since the deterministic entropy of a function is an upper bound on the randomness of a function, a sufficiently large decrease in the deterministic entropy results in a function with a lower randomness, or equivalently increased compressibility.

However, in all the Tic-Tac-Toe tasks the deterministic entropy of the functions that result from both the representations is the same. One representation was better than another in this case because the better representation reduced the randomness of the examples by exposing some regularities that the learning algorithm could utilize to find a compact description for the examples. This method of representation change was called randomness reduction.

6.2 The Reasons for ACR's Effectiveness

The reason that ACR correctly ranks instance representations is its use of compact coding as a basis for ranking representations. The duality between data-compression and generalization explains why the minimum number of bits required to describe the examples is a good indicator of the classification accuracy that can be achieved with a representation. Our experiments show that the correspondence between data-compression and generalization is precise enough that even small differences in accuracy can be detected by determining the minimum codelength for the examples. Also, since the minimum codelength is estimated from all the available examples one does not face the problem of producing a representative testing set. This problem is the reason why fast methods for estimating error-rate are inaccurate, and accurate

methods are time-consuming.

There are two reasons why ACR is faster than methods for ranking representations by estimating error-rates. The first reason is the effectiveness of the two tests for terminating a run. With these tests, ACR can terminate a run by observing only a few samples from the examples. The number of samples required depends on whether the representations being compared are well-suited for the task. Bad representations are identified as being so with a small number of samples. If both representations being compared are well-suited to the task, then the number of samples depends on the amount of difference in the accuracies produced with the different representations. The second reason why ACR is faster is that the ranking obtained in a run usually does not change in multiple runs. As a result, only a few runs are required to have the desired confidence in the ranking. The reason why the ranking does not change in multiple runs appears to be that small differences in accuracies lead to relatively large differences in the codelength. This makes codelength a measure of the predictive accuracy that is robust against sampling errors.

6.3 Limitations of ACR

The main limitations of ACR are the three assumptions required for the tests for terminating a run. The first assumption was required for the rank test, and it stated that once a statistically significant trend is established that codelength increases with sample size, then the trend can be expected to continue. This precludes situations where there is a critical number of examples such that the ability of the learning algorithm to generalize increases significantly for samples of size greater than this number. If the learning algorithm has such a number, and it is known, then the "start" parameter of ACR should be set to this number. The second assumption was that the size of the hypothesis produced by the learning algorithm increases as the number of examples used to form the hypothesis increases. The third assumption was that modifying a hypothesis by incorporating new examples does not restructure the hypothesis completely.

The need for these assumptions and their affect on ACR was discussed in Chapter 3. Another limitation of ACR is that the test for independence of the random-variables marking the errors is an approximate test. Also, prior to using ACR for a particular learning algorithm, one has to devise a coding scheme for the hypotheses that reflects the bias of the learning algorithm. Designing a coding scheme for the hypotheses produced by the learning algorithm requires some thought. However, coding schemes for a number of popular learning algorithms are known; some of these coding schemes were discussed in Chapter 4. In addition, during the development of ACR we made small changes to the coding scheme used to determine the codelength of a decision tree and a preceptron tree. The rankings of the representations did not change for any of the tasks when the coding schemes were modified. This suggests

that ACR is robust against small changes in the coding schemes.

The fact that duality between data-compression and generalize does not hold when hypotheses produced by the learning algorithm has a poor accuracy is also a limitation of ACR because it is based on the duality. Consequently, as was the case for the `tic2` problem for ID3, ACR may rank representations incorrectly when both the representations result in hypotheses with poor predictive accuracy. However, these situations can be detected by observing the rate of increase in the codelength as samples of increasing size are drawn. A very large rate of increase is indicative of the fact that the representation is not well suited for the task. If for all the representations being compared the rate of increase is large, then one knows that none of the representations is well suited for the task. However, precisely determining what qualifies as a large rate of increase requires further research.

Our experience shows that the independence test is a very conservative test for determining whether the codelength can decrease further. This is because the independence of the random-variables marking the errors is only a sufficient condition for determining that the codelength will not decrease. It may be the case that the codelength cannot decrease, and it is not increasing either, even though the random-variables marking the errors are not independent. Efficient tests for identifying this condition will improve the efficiency of ACR.

ACR's use of the learning algorithm may appear to some as one of its limitations. However, we think that ranking of the representations depends on the learning algorithm employed for learning. In the `tic4` task, for rote learning both location-contents and the move-sequence representations were equally good, but for ID3 the move-sequence representation was better than the location-contents representation. In general, different learning algorithms will employ different methods for finding a compact hypothesis, and the features best suited to the compression scheme utilized by the learning algorithm will make the representation that uses those features the best for that learning algorithm. For example, a set of features that make the examples linearly separable, or almost linearly separable, are good for the perceptron tree learning algorithm, but not necessarily for ID3. Therefore, we think that one representation can only be better than another in the context of a particular learning algorithm and that it does not make sense to say that one representation is better than another for all learning algorithms. However, it is possible that further research will uncover techniques that need to observe smaller fractions of the examples than the fraction observed by ACR. Also, it may be possible to categorize learning algorithms based on the compression scheme employed by them, and to rank representations for classes of learning algorithms rather than individual learning algorithms.

It is possible that by analyzing the match between the representation and the compression scheme employed by the learning algorithm, one may be able to rank representations without reference to a particular set of examples. In this dissertation the representation comparison problem was formulated as being one where the goal is to find the representation that enables the learning algorithm to produce the

most accurate hypothesis from the given examples. Such a formulation suffers from a limitation that any ranking based on an unrepresentative set of examples may be incorrect. However, this limitation is shared by all methods for learning from examples, because any inductive reasoning based on unrepresentative observations may be incorrect. The usual method for handling this is to ensure that one has a large sample from which the inferences are drawn. Results from learnability theory can help in determining the number of observations that suffice to ensure that the probability of making incorrect inferences is small. However, these are worst-case bounds and they tend overestimate the number of examples required by a large amount.

6.4 Contributions of This Research

We see this research as making two contributions. First, its scientific contribution is the further explication of the duality between data-compression and generalization. It shows how the correspondence between data-compression and generalization enables one to address issues that arise in learning from examples. ACR's use of compact coding as a representation selection criteria, when viewed in conjunction with the successful use of compact coding as a criterion for hypothesis selection (Quinlan & Rivest, 1989; Segen, 1985; Georgeff & Wallace, 1985; Rissanen, 1989), shows how compact coding provides a unified criterion for addressing issues in learning from examples. Furthermore, the duality between data-compression and generalization suggests that different methods for improving compressibility of the examples will result in different techniques for producing good representations.

The engineering contribution of this research is ACR as a tool for comparing representations. In addition to ranking representations, ACR provides an evaluator for constructive induction algorithms. By using ACR these algorithms can quickly judge whether they are able to produce a good representation for the examples. It can also form a test component of a constructive induction algorithm that searches the space of representations in a generate-and-test manner.

6.5 Directions for Future Research

The research reported in this dissertation can be extended in two directions. The first direction is to investigate the use of compact coding in designing other aspects of a learning system. The second direction is further research on the duality between data-compression and generalization. This section discusses some promising approaches in each of these directions.

6.5.1 Compressibility as a Unified Criterion for Inductive Learning

A designer of a learning system must make a number of choices correctly before a system can learn from examples effectively. This dissertation showed how instance representations can be selected on the basis of the compressibility of the examples. However compact coding can also be used for determining the best learning algorithm for a task and in the simultaneous selection of an instance representation and a learning algorithm for a task.

A learning algorithm can be matched to a task by choosing the algorithm that enables the examples to be encoded most compactly. However, before taking this approach one has to be careful to ensure that the coding schemes for assigning number of bits to hypotheses are not biased to favor the hypotheses produced by one learning algorithm over another. For instance, to determine whether ID3 or PT1 is more suited to a particular task, one can determine which of these two algorithms produces a more compact hypothesis for the examples. However, such a comparison requires that the number of bits used to describe a decision tree is commensurate with the number of bits used to describe a perceptron tree. We think that if provably optimal encodings are used for the different concept description languages then such comparisons are possible. Given such encoding schemes, one can use the two tests used in ACR to estimate the codelength without observing samples of all possible sizes and thereby obtain a quick method for choosing algorithms.

If one can choose algorithms, it is straight-forward to do a simultaneous selection of the combination of the learning algorithm and instance representation best suited for the task. The criterion would be to select the pair that enables the observation to be described most compactly. Again, the two tests used by ACR to terminate a run give a quick method of determining the best combination. Finally, if one can find a computationally efficient method for finding the most compact hypothesis in a concept description language, then one can do simultaneous selection of the best instance representation, concept description language, and the hypothesis. However, recall that algorithmic information is uncomputable in general, so the best possible hypothesis for a set of examples can, in general, never be determined.

In addition to choosing the best representation, compact coding can also result in methods for producing good representations. Determining why a learning algorithm is unable to find a compact hypothesis for the examples can give ideas about improving a representation. Pallago and Haussler (1990) and Matheus (1989) have shown how the lack of compressibility manifests itself in decision trees via the replication problem. They have given heuristic algorithms for improving a representation by removing replication. The duality between data-compression and generalization tells us that this approach can be pursued further by detecting different reasons for the lack of compressibility and devising techniques for removing the defects in the instance representation that cause the lack of compressibility.

Another promising direction is to extend the classification of different kinds of representation change on the basis of how a good representation enhances compress-

ibility. Such a catalog of techniques will provide a designer of representations a set of approaches that can be tried to produce good representations. A similar direction of research is to classify learning algorithms on the basis of the compression scheme employed by them. Given such a classification, one may be able to design instance representations suited to the compression scheme employed by the learning algorithm.

6.5.2 Extending the Duality between Data-compression and Generalization

The duality between data-compression and generalization tells us that the more compactly one describes a set of observations exactly, the better one generalizes from the examples. In the learning tasks considered in this research, no domain specific knowledge was taken into account. However such information is often available. For instance, the task may be to build a system that learns to read handwritten input that consists of programs from a programming language rather than isolated handwritten characters. With this knowledge, one can build a system for reading handwritten programs using the grammar of the programming language and a recognizer for individual characters. Recognizers for individual characters can be learned from examples. If one did not know that the purpose of the recognizer was to produce tokens for a programming language one would need to find a compact hypothesis consistent with all the examples. However, with the knowledge that the recognizer is to be used for recognizing programs, hypotheses that are incorrect on some examples also become possible candidates. This is because the grammar for the language can be used to correct certain errors in recognizing individual characters. Thus, any hypothesis for the recognizer with which one can correctly classify all the examples, using both the recognizer and the grammar, counts as a consistent hypothesis. Such a hypothesis can be more compact than hypotheses required to be consistent with all the examples. These hypotheses should result in better generalization and better performance in reading handwritten programs. In summary, one needs to explore all possible methods that increase the compressibility of the observations because they will result in techniques for improving generalization.

APPENDIX A

KRAFT's INEQUALITY AND ITS IMPLICATION FOR ALGORITHMIC PROBABILITY

Kraft's Inequality

A code $\{c_0, c_1, \ldots, c_{m-1}\}$ is a set of strings from a finite alphabet $\{a_0, a_1, \ldots, a_{j-1}\}$. Each string is called a code-word. A code is called a *prefix code* if no code-word is a prefix of another code-word. If $l_1, l_2, \ldots, l_{m-1}$ are the lengths of the code-words $c_0, c_1, \ldots, c_{m-1}$ respectively, then a code is a prefix code if and only if,

$$\sum_{i=0}^{m-1} j^{-l(c_i)} \leq 1.$$

Where, j is the size of the alphabet and m is the number of code-words.
Proof:
$\Leftarrow$) If a code is a prefix code then $\sum_{i=1}^{m-1} j^{l(c_i)} \leq 1$.

Any prefix code can be arranged as a j-ary tree. The edges from each node of the tree are labeled by the characters of the alphabet. Only the leaves of the tree are valid code-words. The code-word represented by a leaf is given by the concatenation of the characters labeling the edges on the path from the root to the leaf. For example, Figure A.1 represents the prefix code $\{0, 10, 110, 1110\}$ as a tree.

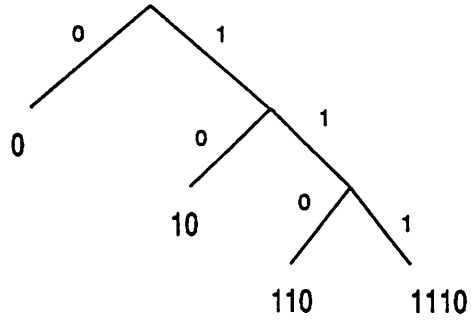


Figure A.1 Tree representation of a prefix code

In a complete j-ary tree there are j^k nodes at depth k. Take a k that is greater than the length of the longest code-word. There are at most j^k nodes at this level. A code-word of length l blocks j^{k-l} of these nodes from being valid code-words. Also, because there is a unique path from root to any node in a tree, no node is blocked by more than 1 code-word. Therefore, the number of blocked nodes is,

$$\sum_{i=0}^{m-1} j^{k-l(c_i)}.$$

Since there are only j^k nodes at level k,

$$\sum_{i=0}^{m-1} j^{k-l(c_i)} \leq j^k.$$

Therefore,

$$\sum_{i=0}^{m-1} j^{-l(c_i)} \leq 1.$$

$\Rightarrow$) If $\sum_{i=0}^{m-1} j^{l(c_i)} \leq 1$, then there exits a prefix code $\{c_0, \ldots, c_{m-1}\}$, where $l(c_i)$ is the code length of the code-word c_i.

Let p be the length of the longest code-word and let there be n_i code-words of length i. Therefore, by counting the code-words in two different ways,

$$\sum_{i=1}^{p} n_i j^{-i} = \sum_{i=0}^{i=m-1} j^{-l(c_i)}$$

equivalently,

$$\sum_{i=1}^{p} n_i j^{-i} \leq 1.$$

However, if a sum of a series is bounded above by 1 then the sum of any part of the series is also bounded above by 1. That is,

$$\begin{aligned} n_1 j^{-1} &\leq 1 \\ n_2 j^{-2} + n_1 j^{-1} &\leq 1 \\ &\vdots \\ n_p j^{-p} + n_{p-1} j^{-(p-1)} + \cdots + n_1 j^{-1} &\leq 1. \end{aligned}$$

Equivalently,

$$\begin{aligned} n_1 &\leq j \\ n_2 &\leq j^2 - n_1 j \\ &\vdots \\ n_p &\leq j^p - n_{p-1} j^{p-1} - \cdots - n_1 j^{l-1}. \end{aligned}$$

The first inequality says that the number of code-words of length 1 is at most than j. In a j-ary tree at level 1 there are j nodes, and these can be used to assign code-words of length 1. The second inequality says that there are at most $j^2 - n_1 j$ code-words of length 2. There are j^2 nodes at level 2 of a j-ary tree, and if n_1 have been used for code-words of length 1, then $n_1 j$ of the j^2 nodes at level 2 are blocked from being valid code-words. That still leaves $j^2 - n_1 j$ nodes which can be used to assign code-words of length 2. A prefix code can be constructed in this manner by following the tree interpretation of the above inequalities.

Algorithmic Probability is a Measure

The use of prefix machines: Algorithmic probability was defined to be,

$$P_U(s) = \sum_{p \in \{0,1\}^* : U(p) = s} 2^{-l(p)}.$$

If U is any universal Turing machine then it is not clear whether $P_U(s)$ will be finite. However, if we restrict U to be such that no program for U is a prefix of another program then we can appeal to Kraft's inequality to show that algorithmic probability exists. Above, Kraft's inequality was proved for finite alphabets, but the result generalizes to countable alphabets also (Rissanen, 1989).

A ensure that no program is a prefix of another consider a universal Turing machine with 3 tapes. A one way, read-only, input tape, one way output tape and two way read-write work tape. The input tape is also called the program tape. The reason why no program is a prefix of another program is that there are no transitions from a halt state to any other state.

Proof that algorithmic probability is a measure: A probability measure on a sample space Ω is a real-valued function of subsets of Ω satisfying three axioms:

1. For every set $A \subset \Omega$, the value of the functions is a non-negative number: $P(A) \geq 0$.

2. For any two disjoint sets A and B, the value of the function for their union $A \cup B$ is equal to the sum of its value for A and its value for B:

$$P(A \cup B) = P(A) + P(B) \text{ provided } A \cap B = \emptyset.$$

3. The value of the function for Ω (as a subset) is equal to 1, that is $P(\Omega) = 1$.

This definition of a probability measure can be obtained from most texts on probability theory, for instance in Chung (1979).

For algorithmic probability, the sample space Ω is the set of programs p for which a prefix-machine U halts. A program of length $l(p)$ has algorithmic probability $2^{-l(p)}$.

This value can never be less than 0. Therefore, algorithmic probability satisfies condition (1). The algorithmic probability of any set of programs, by definition, is equal to the algorithmic probability of each program. Therefore condition (2) is satisfied. However, condition (3), is not true of the for algorithmic probability. However, by Kraft's inequality $P(\Omega) \leq 1$. Here, set functions $\mu(.)$ that satisfy all other conditions required of probability measures, except that $P(\Omega) \leq 1$, are called *measures*. In this sense then, algorithmic probability is a measure.

APPENDIX B

RELATIONSHIP BETWEEN ALGORITHMIC PROBABILITY AND ALGORITHMIC INFORMATION

This appendix proves that the logarithm of algorithmic probability is equal to the algorithmic information up to an additive constant. That is, up to an additive constant,

$$-\log P(s) = K(s)$$

where, $P(s)$ is the algorithmic probability of a string s and $K(s)$ is the algorithmic information of the string s. The main ideas of this proof are from Chaitin (1975). Elementary notions from recursive function theory— for instance at the level covered in Hopcroft and Ullman (1979)— are needed to understand the proof completely.

The proof rests on the construction of a Turing machine M, such that

$$K_M(s) = -\log P(s) + c. \tag{B.1}$$

where, c is a constant that depends only on the machine M and the prefix machine U used for defining algorithmic probability and algorithmic information. The existence of M suffices for proof because the universality of U implies,

$$K(s) \leq K_M(s) + a \tag{B.2}$$

where, a is another constant that depends only on M and U. Substituting (B.1) in (B.2)

$$K(s) \leq -\log P(s) + b \tag{B.3}$$

where b is another constant that depends only on M and U. But, $K(s)$ is the size of the shortest program to compute s. There are many more programs that compute s, consequently,

$$P(s) > 2^{-K(s)}.$$

Equivalently,

$$K(s) \geq -\log P(s). \tag{B.4}$$

Together, B.3 and B.4 prove the theorem.

To show the existence of M, note that the set

$$T = \{\langle s, r\rangle : P(s) > r\}$$

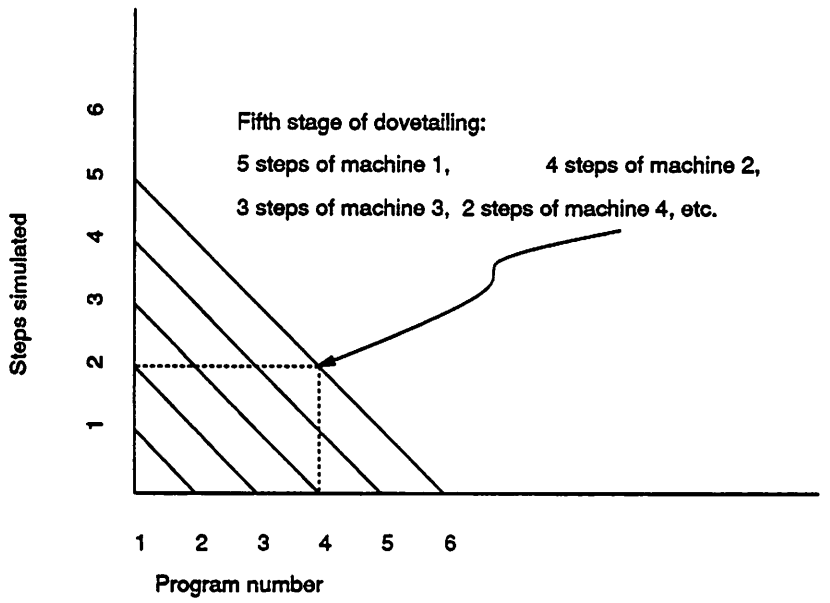


Figure B.1 Dovetailing to show that T is recursively enumerable.

is recursively enumerable, where $\langle .,. \rangle$ is a pairing function. A *dovetailing* argument shows this. Let $p_1, p_2, \ldots$ be an enumeration of programs for U. Consider a Turing machine D that does the following:

Stage 1: Simulate 1 instruction of p_1.
Stage 2: Simulate 2 instructions of p_1, 1 instruction of p_2.
Stage 3: Simulate 3 instructions of p_1, 2 instructions of p_2, 1 instruction of p_3.
$\vdots$
Stage i: Simulate i instructions of p_1, $i-1$ instructions of $p_2, \cdots$, 1 instruction of p_i
$\vdots$

Figure B.1 illustrates this process. If at any stage any of the programs causes U to halt, say it is the program p_j of size n, and say that the output is s, then $\langle s, 2^{-n} \rangle \in T$.

Now, M works as follows. Given a program p, say of length n, on its program tape, M simulates U with program p. If U halts with output s, then by simulating D, M checks whether $\langle s, 2^{-(n-1)} \rangle \in T$. If $\langle s, 2^{-(n-1)} \rangle \in T$ M writes s on its output tape and halts. If M halts with a program of size n, then

$$P(s) > 2^{-(n-1)}.$$

Equivalently,

$$n > -\log P(s) + 1. \tag{B.5}$$

If $P(s)$ is an integer then $\lceil -\log P(s) \rceil = -\log P(s)$ and the smallest n for which (B.5) holds is $n = \lceil -\log P(s) \rceil + 1$. If $P(s)$ is not an integer then too the smallest n

for which (B.5) holds is $n = \lceil -\log P(s) \rceil + 1$. Therefore,

$$K_M(s) = \lceil -\log P(s) \rceil + 1.$$

APPENDIX C

ALGORITHMIC INFORMATION IS A GOOD APPROXIMATION OF THE PRIOR PROBABILITY

This appendix gives a proof for the fact that, except for strings with very small prior probability, algorithmic information is a good approximation to the logarithm of the probability assigned to a string by any computable probability distribution. A computable probability distribution assigns to a string s the probability $\mu(s)$, and there is an algorithm to compute $\mu(s)$ from s. In particular, $K(\mu)$ will denote the size of the shortest program required to compute the string that represents $\mu(.)$. This is possible because if the set of hypotheses is finite then $\mu(s)$ can be represented as a finite string by listing its value for each possible hypothesis.

There are two main steps in the proof. The first step is to show that for every computable distribution μ, there exists a constant c that depends only the Turing machine U used to define algorithmic information, such that:

$$K(x) \leq -\log \mu(x) + K(\mu) + c. \tag{C.1}$$

The second step of the proof is to show by *Markov's inequality* that

$$\mu\{x : K(x) < -\log \mu(x) - r\} < 2^{-r}. \tag{C.2}$$

The relationship (C.2) says that the μ-measure of the set of strings s for which the algorithmic information $K(s)$ is less than $-\log \mu(s)$ by more than r decreases exponentially with r. That is, the prior probability of the strings for which $K(s)$ is less than $-\log \mu(s)$ by more than r is very small. But, (C.1) says that for a given μ, $K(s)$ is always less than the sum of $-\log \mu(s)$ and a fixed constant. Together, (C.1) and (C.2) imply that, except for strings with very small prior probability, $K(s)$ and $-\log \mu(s)$ are equal up to an additive constant. Some ideas in the proof presented here are from Li & Vitányi (1988).

The relationship (C.1) can be demonstrated with the help of a coding technique called *arithmetic coding*. If the implementation details are ignored then arithmetic codes have a property that if a source message a occurs with the probability $p(a)$, then the code length $l(a)$ for that message is given by:

$$l(a) = -\log p(a).$$

Table C.1 Arithmetic code: An example

Symbol	Probability	Partition
a_0	1/8	[0, .125)
a_1	1/8	[.125, .25)
a_2	1/4	[.25, 0.5)
a_3	1/2	[0.5, 1)

The idea in arithmetic coding is to partition the unit interval $[0, 1)$ into sub-intervals and associated each sub-interval with a message. The size of the sub-interval is equal to the probability of occurrence of the message. Table C.1 is an example of such a partitioning of the unit interval. To transmit a message, any binary fraction in the interval corresponding to the message can be transmitted. Since the interval corresponding to the message a has length $p(a)$, $-\log p(a)$ bits are sufficient to represent the interval. Notice that since, $l(a) = -\log p(a)$,

$$\sum_s 2^{-l(a)} = \sum_s 2^{-\log p(s)} = \sum_s p(s) \leq 1.$$

Therefore, arithmetic codes form a prefix code. Rissanen & Langdon (1979), and Witten, Neal & Cleary (1987) provide more details on arithmetic coding. Lelewer and Hirschberg (1987) in their survey article on data-compression give an introduction to arithmetic coding.

A Turing machine T that decodes the arithmetic code p for a string s implies that,

$$K_T(s) = -\log \mu(s).$$

Therefore for a given μ,

$$K(s) \leq -\log \mu(s) + c$$

for some constant c that depends only on T. For the purposes of showing the existence of T, consider the following method of decoding an arithmetic code. First consider a method to produce the arithmetic code for a string s. Given an s, compute $-\log \mu(s)$. This gives the number of bits k in the code-word. Start with these bits set to 0. Then enumerate the strings s, without repetition, in a lexicographical order until s shows up in the enumeration. For every string x that occurs in the enumeration, add $\mu(x)$ to a counter S that records that cumulative probability of the strings observed so far. Each time S becomes greater than 2^{-k}, add 1 to the code-word for s. That is, one tries to identify the appropriate interval of length $\mu(s)$ that corresponds to the string s.

Now T works as follows, given a program p on its program tape, T starts enumerating the strings s on its work tape and constructing the arithmetic code for each

string. If the code for a particular string s is p then T writes s on its output tape. If $K(\mu)$ is the size of the shortest program to compute the string representing μ, then by providing a universal Turing machine with $K(\mu)$, the above procedure can be used to decode the arithmetic code for a string s for any μ. Consequently,

$$K(s) \leq -\log \mu(s) + K(\mu) + c.$$

This completes the first step of the proof.

For the second part of the proof consider,

$$d(s,\mu) = -\log \mu(s) - K(s).$$

So,

$$t(s,\mu) = 2^{d(s,\mu)} = 2^{-\log \mu(s) - K(s)}.$$

The μ-expected value of $t(s,\mu)$ is ,

$$\begin{aligned} & \sum_s \mu(s) 2^{-\log \mu(s) - K(s)} \\ = & \sum_s 2^{-K(s)}. \end{aligned}$$

Since only prefix machines are considered, by Kraft's inequality:

$$\sum_s 2^{-K(s)} \leq 1.$$

Therefore, the μ-expected value of $t(s,\mu)$ is less than or equal to 1.

Now, Markov's inequality says, if Y is a random variable such that $\mu[Y \geq 0] = 1$, then for any fixed $t > 0$ and $a > 0$, $\mu[Y \geq t] \leq E_\mu Y^a / t^a$, where $E_\mu(Y)$ denotes the μ-expected value of Y. Let $a = 1$, $t = 2^k$, and $Y = t(s,\mu)$. Clearly $Y \geq 0$ because one can never get a negative number by exponentiating a positive number. Therefore, by application of Markov's inequality,

$$\mu[2^{-\log \mu(s) - K(s)} \geq 2^k] \leq \frac{E_\mu(2^{-\log \mu(s) - K(s)})}{2^k}.$$

But, $E_\mu(2^{-\log \mu(s) - K(s)}) \leq 1$, therefore,

$$\mu[-\log \mu(s) - K(s) \geq k] \leq 2^{-k}.$$

Equivalently,

$$\mu[K(s) < -\log \mu(s) - k] \leq 2^{-k}.$$

This proves the second part of the theorem.

APPENDIX D

PARAMETERS FOR THE EXPERIMENTS

ACR requires four parameters in addition to the significance levels and the confidence intervals for the various tests. These parameters are, the first sample size considered by ACR, the final sample size considered by ACR, and the amount by which the sample size is incremented at each step. In the experiments reported in Chapter 4, the values of these parameters were set to the same values used in the direct estimation of the accuracies using the hold-out method. Table D gives these values.

Table D.1 Parameter settings for the experiments

Problem name	Start	End	Increment
two-or-more-clumps	25	2500	25
two-clumps	25	2500	25
tic6	10	500	10
tic5	50	2500	50
tic4	100	5000	100
tic3	100	5000	100
tic2	100	5000	100
tic1	200	10000	200
handwriting	20	600	10

To determine an appropriate setting for the maximum sample size, the training and testing sets were combined and the examples were drawn at random, with replacement, from the combined set and presented to the learning algorithm. Each example had a equal probability of being selected. The hypothesis produced by the learning algorithm was used to classify all the examples in the combined set and the accuracy of the hypothesis was recorded. This procedure was continued for samples of increasing sizes, until almost all the examples in the combined set were correctly classified and the accuracy did not change much by changing the sample. That is, the accuracy of the hypothesis produced by the learning algorithm reached an asymptotic value near 100%. The sample size for which this happens represents the sample

size required to learn the concept perfectly. This number was used as the maximum sample size.

REFERENCES

Abramson, N. (1963). *Information Theory and Coding.* McGraw-Hill.

Abu-Mostafa, Y. S. (1986). The complexity of information extraction. *IEEE Transactions on Information Theory, 32*, 513-525.

Abu-Mostafa, Y. S. (1988). Complexity of random problems. In Y. S. Abu-Mostafa (Ed.), *Complexity in Information Theory.* New York: Springer-Verlag.

Arakawa, H., Odaka, K., & Masuda, I. (1978). On-line recognition of handwritten characters—Alphanumerics, Hiranga, Katakana, Kanji. *Proceedings of the Fourth International Joint Conference on Pattern Recognition* (pp. 810-812).

Baum, E. B., & Haussler, D. (1989). What size net gives valid generalization?. *Neural Computation, 1*, 151-160.

Blahut, R. E. (1987). *Principles and Practice of Information Theory.* Addison-Wesley.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1987). Occam's Razor. *Information Processing Letters, 24*, 377-380.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1989). Learnability and the Vapnik-Chervonenkis Dimension. *Journal of the ACM, 36*, 929-965.

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees.* Belmont, CA: Wadsworth International Group.

Callan, J. P., & Utgoff, P. E. (1991). Constructive induction on domain knowledge. *Proceedings, Ninth National Conference on Artificial Intelligence* (pp. 614-619). Anaheim, CA: MIT Press.

Casey, R. G., & Nagy, G. (1984). Decision tree design using a probabilistic model. *IEEE Transactions on Information Theory, 30*, 93-99.

Chaitin, G. J. (1975). A theory of program size formally identical to information theory. *Journal of the ACM, 22*, 329-340.

Chaitin, G. J. (1977). Algorithmic information theory. *IBM Journal of Research and Development, 21*, 350-359,496.

Chatfield, C. (1984). *The Analysis of Time Series: an Introduction (Third Edition).* New York: Chapman and Hall.

Chung, K. L. (1979). *Elementary Probability Theory with Stochastic Processes.* New York: Springer-Verlag.

Cover, T. M. (1965). Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers, 14*, 326-334.

Cover, T. M. (1973). Enumerative Source Coding. *IEEE Transactions on Information Theory, IT-19*, 73-77.

Denker, J., Schwartz, D., Wittner, B., Solla, S., Hopfield, J., Howard, R., & Jackel, L. (1987). Large automatic learning, rule extraction, and generalization. *Complex Systems, 1*, 877-922.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory, IT-21*, 194-203.

Fawcett, T. E. (1991). *A hybrid empirical/analytical method for feature generation* (COINS Technical Report 91-8). Amherst, MA: University of Massachusetts, Department of Computer and Information Science.

Gao, Q., & Li, M. (1989). An application of minimum description length principle to online recognition of handprinted alphanumerals. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* Detroit, Michigan: Morgan Kaufmann.

Georgeff, M. P., & Wallace, C. S. (1985). A general selection criterion for inductive inference. In T. O' Shea (Ed.), *Advances in Artificial Intelligence.* Elsevier Science Publishers, B. V. (North Holland).

Gibbons, J. D. (1971). *Nonparametric Statistical Inference.* New York: MaGraw Hill.

Hollander, M., & Wolfe, D. A. (1973). *Nonparametric Statistical Methods.* New York: John Wiley & Sons.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation.* Reading, Massachusetts: Addison-Wesley.

Jansson, B. (1966). *Random Number Generators.* Stockholm: Victor Pattersons Bokindustri Aktiebolag.

Jaynes, E. T. (1988). How does the brain do plausible reasoning?. In G. J. Erickson, & C. Ray Smith (Eds.), *Maximum-Entropy and Bayesian Methods in Science and Engineering, volume 1.*. Boston: Kulwer Academic Publishers.

Kemeny, J. G. (1953). The use of simplicity in induction. *The Philosopical Review, 62*, 391-408.

Kendall, M. G. (1962). *Rank Correlation Methods.* New York: Hafner Publishing Company, Inc.

Kittler, J. (1986). Feature selection and extraction. In T. Z. Young, & K. S. Fu (Eds.), *Handbook of pattern recognition and image processing.* New York: Academic Press.

Knuth, D. E. (1971). *The Art of Computer Programming, Volume 2.* Reading, MA: Addison-Wesley.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems in Information Transmission, 1,* 1-7.

Kolmogorov, A. N. (1968). Logical basis for information and probability theory. *IEEE Transactions on Information Theory, IT-14,* 662-664.

Lelewer, D. A., & Hirschberg, D. S. (1987). Data compression. *Computing Surveys, 19,* 261-296.

Leung-Yan-Cheong, S. K., & Cover, T. M. (1978). Some equivalences between Shannon entropy and Kolmogorov Complexity. *IEEE Transactions on Information Thoery, IT-24,* 331-338.

Li, M., & Vitányi, P. M. B. (1988). Two decades of applied Kolmogorov complexity. *Proceedings of the Third Annual IEEE Structure in Complexity Theory Conference* (pp. 80-101).

Li, M., & Vitányi, P. M. B. (1989). Inductive reasoning and Kolmogorov complexity (Preliminary Version). *Proceedings of the Fourth Annual IEEE Structure in Complexity Theory Conference* (pp. 165-185).

Maciejowski, J. M. (1979). Model discrimination using an algorithmic information criterion. *Automatica, 15,* 579-593.

Martin-Löf, P. (1966). The definition of random sequences. *Information and Control, 9,* 602-619.

Matheus, C. J., & Rendell, L. A. (1989). Constructive induction on decision trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 645-650). Detroit, Michigan: Morgan Kaufmann.

Matheus, C. J. (1990). Adding domain knowledge to SBL through feature construction. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 803-808). Boston, MA: Morgan Kaufmann.

Mehra, P., Rendell, L. A., & Wah, B. W. (1989). Principled constructive induction. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 651-656). Detroit, Michigan: Morgan Kaufmann.

Michalski, R. S., & Chilausky, R. L. (1980). Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Policy Analysis and Information Systems, 4*, 125-160. (Special issue on knowledge acquisition and induction)

Muggleton, S. (1987). Duce, an oracle based approach to constructive induction. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 287-292). Milan, Italy: Morgan Kaufmann.

Newman, W. M., & Sproull, R. F. (1979). *Principles of interactive computer graphics (second edition).* New York: MacGraw Hill.

Nilsson, N. J. (1965). *Learning machines.* New York: McGraw-Hill.

Pagallo, G., & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning*, 71-99.

Pearl, J. (1978). On the connection between the complexity and credibility of inferred models. *International Journal of General Systems, 4*, 116-126.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81-106.

Quinlan, J. R. (1989). Unknown attribute values in induction. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 164-168). Ithaca, NY: Morgan Kaufmann.

Quinlan, J. R., & Rivest, R. L. (1989). Inferring decision trees using the minimum description length principle. *Information and Computation, 80*, 227-248.

Rendell, L. (1986). A general framework for induction and a study of selective induction. *Machine Learning, 1*, 177-226.

Rendell, L., & Seshu, R. (1990). Learning hard concepts through constructive induction: framework and rationale. *Computational Intelligence, 6*, 247-270.

Rissanen J., & Langdon, G. G. (1979). Arithmetic coding. *IBM Journal of Research and Development, 23*, 149-162.

Rissanen, J. (1989). *Stochastic Complexity in Statistical Inquiry.* New Jersey: World Scientific.

Rumelhart, D. E., Hinton, G. E., & Williams, R.J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, & J. L. McClelland (Eds.),

Parallel distributed processing: Explorations in the microstructure of cognition. Cambridge, MA: MIT Press.

Samuel, A. (1959). Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development, 3*, 211-229.

Saxena, S. (1989). Evaluating alternative instance representations. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 465-468). Ithaca, NY: Morgan Kaufmann.

Saxena, S. (1990). Using description length to evaluate input representations for learning. *Proceedings of the AAAI Spring Symposium on the Theory and Application of Minimal Length Encoding* (pp. 135-139).

Segen, J. (1985). Learning concept descriptions from noisy examples. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 634-636). Los Angeles, CA: Morgan Kaufmann.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal, 27*, 379-423.

Solomonoff, R. J. (1964). A formal theory of inductive inference, Part 1 and Part 2. *Information and Control, 7*, 1-22, 224-254.

Tappert, C. C., Suen, C. Y., & Wakahara, T. (1990). The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 12*, 787-808.

Thorton, C. (1988). A computational model for the data compression metaphor. In T. O' Shea, & V. Sgurev (Eds.), *Artificial Intelligence III: Methodology, Systems, Applications.* Elsevier Science Publishers B. V. (North Holland).

Utgoff, P. E. (1989). Perceptron trees: A case study in hybrid concept representations. *Connection Science, 1*, 377-391.

Vapnik, V. N., & Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications, 16*, 264-80.

Wallace, C. S., & Freeman, P. R. (1989). Estimation and Inference by Compact Coding. *Journal of Royal Statistical Society, Series B, 49*, 240-265.

Watanabe, S. (1985). *Pattern recognition: Human and mechanical.* New York: Wiley & Sons.

Weiss, S. M., & Kulikowski, C. S. (1991). *Computer systems that learn.* Palo Alto: Morgan Kaufmann.

Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM, 30*, 520-540.

Wolff, J. G. (1982). Language acquisition, data compression and generalization. *Language and Communication, 2*, 57-89.

Zvonkin, A. K., & Levin, L. A. (1970). The complexity of finite objects and the development of the concepts of information and randomness by the means of the theory of algorithms.. *Russian Mathematical Surveys, 25*, 83-124.