Machine Learning, 7, 85–99 (1991)
© 1991 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Learning to Predict Non-Deterministically Generated Strings

MOSHE KOPPEL
Department of Math and Computer Science, Bar-Ilan University, 52 900 Ramat-Gan, Israel

Editor: David Haussler

Abstract. In this article we present an algorithm that learns to predict non-deterministically generated strings. The problem of learning to predict non-deterministically generated strings was raised by Dietterich and Michalski (1986). While their objective was to give heuristic techniques that could be used to rapidly and effectively learn to predict a somewhat limited class of strings, our objective is to give an algorithm which, though impractical, is capable of learning to predict a very general class. Our algorithm is meant to provide a general framework within which heuristic techniques can be effectively employed.

Keywords. Prediction, Kolmogorov complexity, minimum description length, learning in the limit

1. Introduction

In order to illustrate what we mean by learning non-deterministically generated strings we consider the case of language acquisition. Let us begin by first considering the classic, and much simpler, case of learning deterministically generated languages. Imagine that we are learning the grammar of a language in the following way. We order the countably infinite set of all strings of words, that is, all potential sentences, and put them to a native speaker with complete knowledge of the language, one at a time. For each potential sentence, the native tells us whether it is a grammatical sentence in the language. We use these answers to form a hypothesis as to what the grammar of the language is and thus to predict the native's answers concerning not-yet-asked potential sentences. We say that the language has been learned if given the answers to a sufficient number of questions, we forever after successfully predict the native's answers, i.e., we can distinguish grammatical sentences from non-grammatical sentences. It is well known (Blum & Blum, 1975; Gold, 1967) that there are algorithms which successfully learn any language provided that some (any) bound on the computational resources (i.e., time) required for determining "grammaticalness" in the language is known in advance.

Let us now extend this notion of learning to the case of non-deterministically generated languages. Suppose that the native doesn't have complete knowledge of his language so that he doesn't know the answer to every question. Let us suppose further that he is proud and rather than admit that he doesn't know an answer he guesses (say, by surreptitiously tossing a coin). We wish to use his answers to learn what he knows about the language.

Let us call the native's knowledge (yes, no, don't know) regarding a particular query, the "computed response" and let us call the answer (yes, no) he gives us, the "actual response."

Then we wish to use the native's actual responses to previous queries in order to predict his computed resonse to a non-yet-asked query. (Strictly speaking, then, the "non-deterministically generated language" which we wish to learn is the language known by the native rather than the actual language of his people.)

The difficulty, of course, lies in the fact that a guess "looks" exactly like an honest answer. Nevertheless, we will give an algorithm which can successfully learn any native's knowledge provided that the native's guesses are indeed random and that, as in the deterministic case, some bound on the computational resources required to compute grammaticalness (yes, no, or don't know) in the language known by the native is known in advance.

To further motivate the discussion let us consider another example. Suppose we wish to determine under what circumstances some physical event takes place. Let us assume that there is some natural ordering of a countably infinite set of experiments which can be run to test for the event. From the results of these experiments we form hypotheses as to the circumstances under which the event takes place. Now suppose that the situation is complicated by the fact that it is possible that in some of the experiments the parameters which we control do not determine the outcome, i.e., that there is a "hidden variable" which is independent of the controlled parameters and that the results of these experiments depend on this hidden variable. If it is further assumed that the hidden variable is two-valued and each value occurs with probability 1/2, it is not difficult to see that the learning problem here is isomorphic to that of the language example and that therefore our algorithm works equally well for either example.

The outline of the article is as follows: After formalizing the notion of non-deterministically generated strings, we define the concept of "learning" non-deterministically generated strings by extending the classical definitions of learning-in-the-limit (Gold, 1967) and learning with probability 1 (Osherson, Stob, & Weinstein, 1985; Wexler & Culicover, 1985). Our definition extends learning to non-computable strings.

We briefly review aspects of the theory of program-length-complexity (Chaitin, 1975; Kolmogoroff, 1965; Solomonoff, 1964) in order to introduce a version of information-compression which distinguishes structure from randomness in binary strings. This compression method facilitates the construction of an algorithm which solves the learning problem illustrated above.

Finally, we work the algorithm on some artificial examples in order to illustrate some significant features of the "logic of discovery" which are reflected by the algorithm.

2. Formalizing the problem

Unless otherwise stated all strings in this article are binary. We will use the following notation. Let S be a finite or infinite string. Then $|S|$ is the length of S, S_n is the n^{th} bit of S and S^n is the initial segment of S with length n. For two strings, S and T, $S \leq T$ means that S is an intial segment of T.

Let us consider a generalization of the examples given above. Imagine that the computed response (yes, no, don't know) to a query (potential sentence, experiment) might depend on all the previous actual responses (including guesses). Then we can formalize the computed response as a computable function f which maps each finite binary string (i.e., the previous actual responses) to an element of the set {0, 1, c} (where c stands for "coin" and represents the "don't know" case discussed above). We call such an f a *non-deterministic bit generator* (NDBG). Observe that the illustrations in the introduction are both special cases of NDBGs in which f depends only on the length of the given string but not on its bits. The results of this paper obtain for the general case in which f depends on any or all previous bits (responses).

Another special case of an NDBG is one in which f maps each string to either 0 or 1 (but not c). We call such an f a *deterministic bit generator* (DBG). A DBG f generates a sequence F defined by the equations $F_1 = f(\phi)$ and $F_{n+1} = f(F^n)$. F represents the infinite sequence of responses determined by f.

For example, let f_0 be a DBG such that

$$f_0(x) = \begin{cases} 0 & \text{if } |x| + 1 \text{ is composite} \\ 1 & \text{if } |x| + 1 \text{ is prime} \end{cases}$$

Then the associated sequence F_0 is the characteristic string of the primes.

In general, however, an NDBG might generate an uncountable number of different sequences depending on how the non-determinism is resolved. Let D be a binary string which we call a *decision string* which is used to resolve the non-determinism of f in the following way: we generate the sequence $F_1, F_2, \ldots$ until the first c is reached. Then c is replaced with D_1. We continue generating the sequence in this way, each time replacing the i^{th} c with D_i until the $|D| + 1^{st}$ c is reached (that is, until D is all used up) and then stop. Thus for each decision string D, the NDBG f generates some (finite or infinite) sequence. Let us call this sequence F(D) and refer to F as the *sequence-function* associated with the NDBG f.

Let us consider some examples of NDBGs and their associated sequence functions.

1. Let f_1 be an NDBG such that

$$f_1(S) = \begin{cases} 1 & \text{if } |S| \equiv 3 \pmod 4 \\ c & \text{otherwise} \end{cases}$$

Then the associated sequence-function F_1 is such that

$$F_1(a_1a_2a_3a_4a_5a_6\ldots) = a_1a_2a_31a_4a_5a_61\ldots.$$

That is, f_1 generates strings in which every fourth bit is 1 and all the other bits are determined by coin-toss.

2. Let f_2 be an NDBG such that

$$f_2(x) = \begin{cases} c & \text{if } |x| \text{ is even} \\ 0 & \text{if } |x| \text{ is odd and the last bit of x is 0} \\ 1 & \text{if } |x| \text{ is odd and the last bit of x is 1} \end{cases}$$

Then the associated sequence-function F_2 is such that

$$F_2(a_1a_2\ldots a_n) = a_1a_1a_2a_2 \ldots a_na_n.$$

3. Finally, let P be some computable predicate and let g: N $\rightarrow$ N be some computable function, and let $f_3(x)$ be some NDBG such that

$$f_3(x) = \begin{cases} P(|x| + 1) & \text{if } P(|x| + 1) \text{ can be computed within } g(|x|) \text{ steps} \\ c & \text{otherwise} \end{cases}$$

Then the associated sequence F_3 is such that $F_3(D)$ is the characteristic string of P with P(n) replaced by D_i if n is the i^{th} number for which P(n) fails to be computed in g(n) steps or less.

Now let us consider some of the properties of a sequence-function F:

i. A *partial function* on the finite binary strings is a function which is defined on a subset of the finite binary strings. A function which is defined for all finite binary strings is said to be *total*. For any total NDBG f, the associated sequence function F is total.
ii. A function g from finite strings to (finite or infinite) strings is a *process* if (1) [$D' > D$ and g(D) is finite and g(D') is defined] $\Rightarrow$ $g(D') > g(D)$ and (2) [$D' > D$ and g(D) is infinite] $\Rightarrow$ $g(D') = g(D)$. Any sequence function F is a process. Observe that if a process is defined for finite strings then it is implicitly defined for infinite strings since for any infinite D, $f(D) = \lim n \rightarrow \infty\ f(D^n)$. Note that $\lim n \rightarrow \infty$ $f(D^n)$ must be infinite.
iii. A total function g is *injective* if $g(D') \geq g(D) \Rightarrow D' \geq D$. A total function g is *strictly injective* if for every D, $g(D \cdot 1) \geq g(D) \cdot 1$ and $g(D \cdot 0) \geq g(D) \cdot 0$. Any sequence function F is strictly injective.
iv. A partial function g is *partially computable* if there is a program (Turing machine) M such that for all x, either M(x) and g(x) are undefined or $M(x) = g(x)$. g is *computable* if it is both partially computable and total. If the NDBG f is computable then the associated sequence function F is computable.

These properties are the only ones which we need. Formally, then, we say that F is a sequence function if it is a computable, strictly injective process.

Obviously each NDBG f has a unique associated sequence-function F. For the converse, let F be a sequence-function and say that a string S is *generated* by F if for some D, $S \leq F(D)$. Let D_0 be the shortest D such that $S \leq F(D)$. By the strict injectivity of F, D_0 is uniquely defined. Then F is associated with any NDBG f such that

$$f(S) = \begin{cases} 0 & \text{if } F(D_0) \geq S \cdot 0 \\ 1 & \text{if } F(D_0) \geq S \cdot 1 \\ c & \text{if } F(D_0) = S \end{cases}$$

We say that any such NDBG f is associated with F. Observe that F does not determine f(S) for strings S which are not generated by F.

3. Learning

In this section we review the formal definition of learning-in-the-limit (Blum & Blum, 1975; Gold, 1967) and extend this definition to a version of probabilistic learning which is applicable to NDBGs.

To make matters simple, let us imagine a two-person game in which one player is called the "learner" and the other is called the "native." The native provides the learner with the first n bits of a binary string and the learner then uses some algorithm to form a hypothesis regarding how these n bits were computed and accordingly guesses what the next bit will be. The native then provides the learner with the next bit of the string, thus confirming or contradicting the learner's guess. The learner then uses his algorithm to predict the next bit and the game continues. If from some stage on, the learner's predictions are always correct we say that the learner has learned to predict the native's string. More formally, identifying the learner with the algorithm A which he uses to make his predictions, we have:

Definition. An algorithm A is said to *learn to predict* the infinite string S if for all sufficiently large n, $A(S^n) = S_{n+1}$

Of course, any algorithm will learn *some* string. What makes some learning algorithms more useful than others is their ability to learn many strings.

Definition. An algorithm A is said to *learn to predict the set* of infinite strings β if A learns to predict every $S \in \beta$. If there is an algorithm which learns to predict β, then β is said to be *learnable.*

An important concept in the theory of learning is that of "g-boundedness" which we define now informally. If g: $N \to N$ is a computable function, then a string is said to be *g-bounded* if its initial segments of length n can be computed in at most g(n) steps. We say that the sequence-function F is g-bounded if, given any infinite D, the initial segments of F(D) of length n can be computed in g(n) steps. We say that an NDBG is g-bounded if the associated sequence-function is g-bounded. (This definition is informal because formally g-boundedness is a function of the method of computation being used. Later we will give the formal definition.)

One classic example of a learnable set of strings is the set of g-bounded strings for some g. The algorithm which learns it first orders all DBGs and then, given some initial segment of S, searches through them in order until it finds a DBG which generates that segment within the allotted number of steps and predicts accordingly. Since any DBG which does not generate S will eventually fail for some initial segment, this algorithm will eventually settle on a DBG which does generate S. Once it does so, all its predictions will be correct. (Note that the bound g is necessary in order that the search through the programs not get trapped in a non-halting program.)

Observe that due to the enumerative character of this algorithm its run-time renders it useless for practical purposes. Nevertheless, this algorithm is enormously helpful as a framework within which heuristic techniques can be employed.

Let us now extend our learning game so that the native might sometimes be tossing a coin. The learner is given a string and uses it to form a hypothesis as to how the string has been generated and thus to predict whether the next bit will definitely be 1, definitely be 0, or will be determined by coin-toss. The native then computes whether the next bit is 0 or 1 or is to be determined by coin-toss (where the computation might depend on his previous responses but does not depend on the learner's predictions). Then, tossing a coin if necessary, he gives the learner the next bit without revealing if it was produced deterministically or by coin-toss. The learner then applies his algorithm to attempt to predict the next bit and the game continues. Note that unlike the standard case of learning, here we do not expect the learner to predict the next bit (since we cannot expect him to predict the outcome of a coin-toss) but rather to predict whether it is determined computationally and if so what it will be. Thus we have:

Definition. If F is the sequence-function associated with the NDBG f, and D is an infinite decision string, then we say that the algorithm A *learns* f *with* D if for all sufficiently large n, $A(F^n(D)) = f(F^n(D))$.

Now, of course, some algorithm might learn some NDBG with some decision strings but not with others. We say that a set of infinite strings $\mathcal{K}$ is of measure 1 if with probability 1 a string produced by coin tossing is in $\mathcal{K}$. Then we have

Definition. An algorithm A *learns the* NDBG f *with probability 1* if for some set $\mathcal{K}$ of measure 1, A learns f with all $D \in \mathcal{K}$.

Finally, we have:

Definition. An algorithm A *learns the set of* NDBGs $\mathcal{N}$ *with probability* 1 if for each $f \in \mathcal{N}$, A learns f with probability 1.

Our main result is that for any computable g there is an algorithm which learns the set of g-bounded NDBGs with probability 1.

4. Compression

To lay the groundwork for the algorithm we need to first understand why the learning problem for NDBGs is much harder than the learning problem for DBGs.

Suppose, for example, that in some "learner-native game" the learner knows in advance that the native is using a g-bounded DBG to generate the string and suppose further that the native is generating the string 1 1 1 1 1 . . . Since any DBG which generates anything other than 1 1 1 1 1 . . . will ultimately be contradicted by the native, the algorithm which searches all relevant DBGs exhaustively will eventually settle on the right one.

This is not the case where the native is not restricted to DBGs. The string 1 1 1 1 1 . . . might be generated deterministically, or some or all of the bits might be generated by (some rather fortuitous) coin-tossing. Thus the method of exhaustive search will fail because NDBGs other than the one actually being used might never be falsified by the native. For example, if the learner always predicted that the next bit will be determined by coin-toss no sequence of bits provided by the native can ever force him to change his theory.

The solution to this difficulty lies in the fact that with probability 1 a coin-tossing sequence is "random," i.e., patternless. Therefore, any pattern present in a long string is probably "programmed in" rather than a result of fortuitous coin-tossing. We will show that any pattern which persists as more bits are generated is, with probability approaching 1, not accidental. Moreover, any deterministic features of an NDBG which is generating a string will become apparent in the form of patterns in that string. Thus an algorithm which distinguishes pattern from randomness will, with probability 1, eventually always predict correctly.

We draw upon the theory of program-length complexity to give a formalization of "pattern" which is appropriate for solving our problem. The program-length complexity of a string S was originally defined by Kolmogoroff (1965), Solomonoff (1964) and Chaitin (1975) as the length of the shortest description of S, where by a "description" of S we mean an input to a universal Turing machine which results in S as output. We will use a variant of the original definition described in Koppel and Atlan (in press).

We begin by defining a universal Turing machine.

Definition. Let U be a Turing machine with two input tapes. U is said to be *universal* if for every partially computable process F (and for no other functions) there is an input z such that for every w, U(z, w) = F(w).

(This definition is slightly different than the standard one in which the word "function" appears in place of "process." For a simple construction of this non-standard variety see Schnorr (1973).)

We call the first input to a universal Turing machine a "program" and we call the second input the "data." If U(z, x) = F(x) for all x we say that z computes F and we call z an *F-program*. Note that for every partially computable process F, there are infinitely many F-programs.

We say that z is a *total program* if it is an F-program for a total process F.

Definition. The *complexity* of a finite string S relative to U is

$$H(S) = \min\{|z| + |x| \mid U(z, x) \geq S \text{ and } z \text{ is total}\}.$$

H(S) is the length of the shortest description of S in terms of a total program and data to that program. (In other contexts, we require also that z be "self-delimiting" but since this requirement is irrelevant here we dispense with it for the sake of simplicity.)

Earlier versions of program-length complexity (Chaitin, 1975; Kolmogoroff, 1965; Solomonoff, 1964) were defined in terms of a single input. The distinction between program and data which we apply in the definition reflects the distinction between structure and randomness in a string and is thus useful for predicting non-deterministically generated strings.

For an infinite string x, let x_n^F be the shortest prefix of x such that $F(x_n^F) \geq F^n(x)$. That is, x_n^F is just long enough such that $F(x_n^F)$ is at least n bits long.

The critical concept is the following:

Definition. F is a *compression process* for the infinite string y if there exist x and c such that $F(x) = y$ and for all n, $|x_n^F| \leq H(y^n) + c$.

That is, F is a compression process for y if F together with appropriate input constitute a minimal (or nearly minimal) description of y. Roughly, the idea is that F uses any patterns in y to reconstruct it from as small an x as possible. Whether a process is a compression process for some string does not depend on the choice of U which is used to define H (Koppel & Atlan, in press).

The concept of compression programs can be used to give a neat definition of randomness.

Definition. Let $I(x) = x$ for all strings x. An infinite string y is *random* if I is a compression process for y.

Since $|x_n^I| = n$, x is random if and only if there exists c such that for all n, $H(x^n) \geq n - c$. This is a variant of the definitions of randomness given in Levin (1973) and Schnorr (1973).

Theorem 1. If F *is a sequence-function and* x *is random then* F *is a compression process for* F(x).

Proof. We will show that if F is not a compression process for F(x) then for all c there exists n such that $H(x^n) < n - c$ which contradicts the randomness of x. Let $F(x) = y$. If F is not a compression process for F(x) then for all c there exists n and strings z(n) and w(n) such that $U(z(n), w(n)) \geq y^n$ and z(n) is total and $|z(n)| + |w(n)| < |x_n^F| - c$.

For any finite string y let $F'(y) =$ the longest v such that $F(v) \leq y$. (Think of F′ as a sort of inverse of F.) Since F is strictly injective, F′ is a computable function. Let $z_{F'}$ be a program which computes the composition of the function computed by z with the function F′. Then there is a constant c′ such that for any total program z, $z_{F'}$ is a total programs such that $|z_{F'}| \leq |z| + c'$ and for any string w, $U(z_{F'}, w) = F'(U(z, w))$. Then for all n, $U(z(n)_{F'}, w(n)) \geq x_n^F$ and therefore for all n, $H(x_n^F) \leq |z(n)_{F'}| + |w(n)| \leq |z(n)| + |w(n)| + c'$. But since for any c we can choose n such that $|z(n)| + |w(n)| \leq x_n^F - (c + c')$ it follows that for any c there exists n such that $H(x_n^F) < |x_n^F| - c$ which contradicts the randomness of x.

Since it has been shown (Koppel & Atlan, in press; Levin, 1973) that with probability 1 a sequence generated by coin-tossing is random it follows that

Theorem 2. If F *is a sequence-function and* D *is an infinite binary sequence generated by coin-tossing, then with probability* 1, F *is a compression process for* F(D).

This theorem has direct bearing on our problem. It tells us that if we are receving bits of an infinite string S, which is being generated by some NDBG f, then with probability 1 the sequence-function F associated with f is a compression process for S.

Of course, there are many different compression processes for any given S, so that the fact that F is a compression process for S does not uniquely define F. Nevertheless, the following theorem tells us that for purposes of prediction, all compression programs are equivalent in the limit.

Convergence Theorem. If F_1 *and* F_2 *(associated with the NDBGs* f_1 *and* f_2*, respectively) are both injective compression processes for the infinite string* y*, then for all sufficiently large* n, $f_1(y^n) = f_2(y^n)$.

The Convergence Theorem is proved in Koppel & Atlan (in press).

Let us sum up what we have so far. From Theorem 1 we know that if an NDBG f is used together with a random decision string D to generate a string S then the sequence-function F associated with f is an injective compression process for S (= F(D)). Moreover, from the Convergence Theorem all injective compression processes eventually yield the same predictions. Therefore, if we can find *any* compression process for S we can learn to predict it.

5. The algorithm

In this section we show how to find a compression process for a given string. First we formally define what it means to "find a compression process."

Definition. An algorithm B which maps finite binary strings to programs is said to *find a compression process* for the infinite string y if for all sufficiently large n, $B(y^n) = z$ where z computes a compression process for y.

We will give an algorithm which, given any string S = F(D) (where F is a g-bounded sequence-function and D is a random infinite string), finds a compression process for S. First we need to formally define "g-bounded."

Definition. Let g: $N \rightarrow N$ be any computable function. We say that a program z is *g-bounded in* U if for any infinite string x, the computation of U(z, x) results in at least n bits of output within g(n) steps of computation. We say that the sequence-function F is g-bounded in U if there is some F-program z such that z is g-bounded in U. We say that an NDBG is g-bounded in U if the associated sequence-function is g-bounded.

Borrowing loosely from Bennett (1988), we have:

Definition. A string S is *g-shallow in* U if there is a g-bounded strictly injective compression process for S.

Theorem 4. Let g: N → N *be some computable function. Then there exists an algorithm which finds a strictly injective compression process for any string which is g-shallow in* U.

Main Theorem. Let g: N → N *be some computable function. There exists an algorithm which learns to predict with probability* 1 *any NDBG which is g-bounded in* U.

Proof of Main Theorem from Theorem 4. Let B be an algorithm which maps strings to programs such that it finds a strictly injective compression process for any g-shallow string. Let S be some finite string and let B(S) be the program z. Now let the algorithm A be as follows. Given B(S) = z, compute U(z, x) on all strings x in lexicographic order until the first string, say x_0, is found such that $U(z, x_0) \geq S$ and $U(z, x_0)$ can be computed in less than $g(|S|)$ steps. Then let

$$A(S) = \begin{cases} 0 & \text{if } U(z, x_0) \geq S \cdot 0 \\ 1 & \text{if } U(z, x_0) \geq S \cdot 1 \\ c & \text{if } U(z, x_0) = S \end{cases}$$

Observe that if B(S) computes a sequence-function, say H, and h is an NDBG associated with H, then A(S) = h(S).

Now let f be some g-bounded NDBG with associated sequence-function F and let D be some random string. Then, by Theorem 1, F is a g-bounded strictly injective compression process for F(D) so that F(D) is g-shallow. Then for all sufficiently large i, $B(F^i(D))$ computes a strictly injective compression process, say F′, for F(D). Let f′ be any NDBG associated with F′. Then for all sufficiently large i, $A(F^i(D)) = f'(F^i(D))$. But by the Convergence Theorem, for all sufficiently large i, $f'(F^i(D)) = f(F^i(D))$. Thus for any random D, A learns to predict F(D), i.e., A learns to predict f with probability 1. QED

Finally, to prove Theorem 4 we simply give the algorithm.

We begin by defining weakened versions of g-boundedness and strict injectivity for which programs can be easily checked.

Definition. The program z is *g-bounded until n* if for all $i \leq n$ and all infinite strings x, U(z, x) prints i bits of output within g(i) steps.

Definition. The program z is *strictly injective until n* if for all x, $U^n(z, x \cdot 0) = (U(z, x) \cdot 0)^n$ and $U^n(z, x \cdot 1) = (U(z, x) \cdot 1)^n$

The first step in the algorithm is to compute

$$H_g(S) = \min \{|z| + |x| \mid U(z, x) \geq S \text{ and z is g-bounded until } |S|\}.$$

This computation can be carried out (inefficiently) by exhaustively searching through all pairs (z, x) in order of increasing |z| + |x| until a pair is found such that U(z, x) ≥ S and U(z, x) can be computed in at most g(|S|) steps.

Next, let $A_c(S)$ be the shortest program z such that

1. z is g-bounded until $|S|$, and
2. z is strictly injective until $|S|$, and
3. there exists x such that
   (i) $U(z, x) \geq S$, and
   (ii) for all $n \leq |S|$, $|z| + |x^z_n| \leq H_g(S^n) + c$

For any c, $A_c(S)$ can be computed by exhaustively searching through all pairs (z, x) satisfying $|z| + |x| \leq H_g(S) + c$ in order of increasing $|z|$ until a satisfactory pair is found. Checking each pair requires at most $g(|S|)$ steps.

Now compute $A_c(S)$ for $c = 0, 1, 2, \ldots$ until a value of c, say c^*, is found such that $A_c(S)$ is non-empty. Finally, let $A(S) = A_{c^*}(S)$.

Claim: if y is a g-shallow infinite string then for all sufficiently large n, $A(y^n)$ is a strictly injective compression process for y.

Proof of Claim: For any strictly injective compression process for y, F, there is some unique x such that $F(x) = y$ and some smallest c, call it c(F), such that for all i, $|x^F_i| \leq H_g(y^i) + c$. There is some particular g-bounded strictly injective compression process for y, F′, such that for any g-bounded strictly injective compression process for y, $c(F') \leq c(F)$.

Since F′ is g-bounded and strictly injective it follows that for all n, there is some smallest F′-program z such that z satisfies conditions (1) and (2) in the definition of $A_{c(F')}(y^n)$ and by the definition of c(F′), z satisfies (3) as well. Moreover, any string shorter than z which is not an F′-program must, by the minimality of c(F′), fail to satisfy the second part of condition (3) for some n. Also, any string shorter than z which is not g-bounded or not strictly injective must eventually fail to satisfy conditions (1) and (2), respectively. Therefore, for all sufficiently large n, $A_{c(F')}(y^n) = z$. From the minimality of c(F′) it also follows that for all sufficiently large n, $A_c(y^n)$ is empty for all $c < c(F')$. But then for all sufficiently large n, $A(y^n) = A_{c(F')}(y^n) = z$ which satisfies the requirements of the theorem.

6. Examples

We consider here two of the examples mentioned in Section 2 in order to clarify the above discussion and to illustrate some points about learning in general.

Example 1. Recall that f_1 is the NDBG such that

$$f_1(S) = \begin{cases} 1 & \text{if } |S| \equiv 3 \pmod 4 \\ c & \text{otherwise} \end{cases}$$

The associated sequence-function F_1 is such that $F_1(a_1a_2a_3a_4a_5a_6\ldots) = a_1a_2a_31a_4a_5a_6\,1\ldots$. That is, f_1 generates strings in which every fourth bit is 1 and all the other bits are determined by coin-toss. Now let y be generated by f_1 with some random coin-tossing sequence which happens to begin with 38 consecutive 1's. Thus y begins with 50 consecutive 1's. We will apply our algorithm to this string.

For heuristic purposes we will narrow our attention in advance to just five programs, the lengths of which we assign arbitrarily. Assume that each of these programs is g-bounded.

Let P be a program of length 1 which prints the data. Then $|x_n^P| = n$.

Let N be a program of length 8 which prints an infinite sequence of 1's without regard to the data. Then $|x_n^N| = 0$.

Let M be a program of size 16 which computes the process F_1 described above. Then $|x_n^M| = 3n/4$.

Let R be a program of length 20 which prints 50 1's and then prints the data. Then $|x_n^R| = \max(0, n - 50)$.

Let K be a program of length 25 which prints 48 1's and then behaves like M. Then $|x_n^K| = \max(0, 3(n - 48)/4)$.

In Table 1 we indicate for each n the value of $|z| + |x_n^z|$ where z ranges over the above-mentioned programs. For each n, the smallest of these values is $H_g(y^n)$. The figures in parentheses are $|z| + |x_n^z| - H_g(y^n)$.

In order to compute $B(y^k)$ find the column for which the largest number in parentheses for the rows $n = 1, \ldots, k$ is smallest. That number is $c*$ and the program which is associated with that column is $B(y^k)$.

Table 1. Finding a compression program for the string y generated by F_1.

n	$\|P\| + \|x_n^P\|$	$\|N\| + \|x_n^N\|$	$\|M\| + \|x_n^M\|$	$\|R\| + \|x_n^R\|$	$\|K\| + \|x_n^K\|$	c*	$B(y^n)$
1	2 (0)	8 (6)	17 (15)	20 (18)	25 (23)	0	P
2	3 (0)	8 (5)	18 (16)	20 (17)	25 (22)	0	P
⋮							
7	8 (0)	8 (0)	22 (14)	20 (12)	25 (17)	0	P
8	9 (1)	8 (0)	22 (14)	20 (12)	25 (17)	1	P
⋮							
13	14 (6)	8 (0)	27 (19)	20 (12)	25 (17)	6	P
14	15 (7)	8 (0)	27 (19)	20 (12)	25 (17)	6	N
⋮							
50	51 (43)	8 (0)	54 (46)	20 (12)	27 (19)	6	N
51	52 (31)	-	55 (34)	21 (0)	28 (7)	18	R
⋮							
79	80 (31)	-	76 (27)	49 (0)	49 (0)	18	R
80	81 (32)	-	76 (27)	50 (1)	49 (0)	18	R
⋮							
151	152 (49)	-	130 (27)	121 (18)	103 (0)	18	R
152	153 (50)	-	130 (27)	122 (19)	103 (0)	19	R
⋮							
171	172 (54)	-	145 (27)	141 (23)	118 (0)	23	R
172	173 (55)	-	145 (27)	142 (24)	118 (0)	23	K
⋮							⋮

Several points should be noted. First, note that structure in a string is discovered in stages.

For $n \leq 13$, $B(y^n) = P$ which means that having observed up 13 bits of y, B regards it as a random string.

For $14 \leq n \leq 50$, $B(y^n) = N$ which means that having observed from 14 to 50 bits of y, B assumes that y is a string consisting solely of 1's.

For $51 \leq n \leq 171$, $B(y^n) = R$ which means that having discovered that y does not consist solely of 1's, B regards y as being random after the initial 50 1's.

Finally, for $n \geq 172$, $B(y^n) = K$ which means that B settles on the idea that after 48 1's, y is random except that every fourth bit is a 1.

Second, note that B changes its hypotheses for one of two reasons. The obvious one is illustrated at $n = 51$ where the program N is dropped as a description of y because N is contradicted by the 51st bit of y being 0. The less obvious reason is illustrated at $n = 14$ and $n = 172$ where structure which was not previously apparent (in the sense that the associated program was not economical) becomes apparent and therefore one program is replaced by a more deterministic one.

Finally, note that the program K which B finally settles on does not compute the sequence-function F_1 which was actually used in generating y. Call the sequence-function computed by the program K, F', and let f' be an NDBG associated with F'. Then the critical point here is that for all sufficiently large n, $f'(y^n) = f_1(y^n)$.

Example 2. In order to further illustrate the first two points above we offer one more illustration of the algorithm B. Recall that f_0 is the DBG such that

$$f_0(S) = \begin{cases} 1 & \text{if } |S| + 1 \text{ is prime} \\ 0 & \text{otherwise} \end{cases}$$

f_0 generates only a single string, namely the characteristic string of the primes, $y = 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0 \ldots$ We will apply our algorithm to this string.

Again for heuristic purposes, let us restrict our attention to a narrow list of programs with arbitrarily assigned lengths. Assume that all these programs are g-bounded.

Let P be as in the first example.

Let E be a program of length 20 which maps $a_1a_2a_3\ldots$ to $a_1 1 a_2 0 a_3 0 a_4 0\ldots$ That is E ensures that all even bits (after 2) are 0 but otherwise prints the data. Then $|x_n^E| = n/2$.

Let T be a program of length 30 which ensures that all even bits (above 2) and all multiple-of-three bits (above 3) are 0 and otherwise prints the data. Then

$$|x_n^T| = \begin{cases} \lfloor n/3 \rfloor & \text{if } n \equiv 3 \text{ or } 4 \pmod 6 \\ \lceil n/3 \rceil & \text{otherwise} \end{cases}$$

Finally, let PM be a program of length 100 which prints the characteristic string of the primes regardless of the data. Then $|x_n^{PM}| = 0$.

Table 2 indicates $B(y^n)$ for $n = 1, 2, \ldots.$

Table 2. Finding a compression program for y generated by F_0.

n	$\lvert P\rvert + \lvert x_n^P\rvert$	$\lvert E\rvert + \lvert x_n^E\rvert$	$\lvert T\rvert + \lvert x_n^T\rvert$	$\lvert PM\rvert + \lvert x_n^{PM}\rvert$	c*	$B(y^n)$
1	2 (0)	21 (19)	31 (29)	100 (98)	0	P
2	3 (0)	21 (18)	31 (28)	100 (97)	0	P
⋮						
73	74 (19)	57 (2)	55 (0)	100 (45)	19	P
74	75 (20)	57 (2)	55 (0)	100 (45)	19	E
⋮						
218	219 (119)	129 (29)	103 (3)	100 (0)	29	E
219	220 (120)	130 (30)	103 (3)	100 (0)	29	T
⋮						
504	505 (405)	272 (172)	198 (98)	100 (0)	98	T
505	506 (406)	273 (173)	199 (99)	100 (0)	98	PM
⋮						⋮

In this example we see clearly how the algorithm B progressively finds more structure, proceeding from P to E to T to PM. As it does, the prediction algorithm A makes more precise predictions. For some small n, A predicts incorrectly that the bit succeeding y^n will be determined by coin-toss, while for all $n > 504$, A predicts $f_0(y^n)$ perfectly.

7. Conclusion

The algorithm A can in principle learn to predict an extremely large class of non-deterministically generated sequences. Nevertheless, we do not suggest our algorithm as a realistic model of natural learning. This is because any ordering of programs assigns almost all interesting programs numbers so large that an exhaustive search for such programs requires a prohibitive amount of time. Therefore this sort of algorithm must be used in conjunction with heuristic techniques such as those suggested in Dietterich & Michalski (1986).

Nevertheless this algorithm reflects several very interesting features of natural learning and of the "logic of discovery."

Hypotheses are checked in order of economy. Observe that descriptions of a string in terms of a program z and data x are checked in order of $|z| + |x|$, i.e., in order of simplicity of the description. The effectiveness of the algorithm illustrates the operational value of Ockham's razor.

Hypotheses which, upon the receipt of new information, cease to be economical are rejected even if they are not falsified by the new information. The algorithm B might, upon receiving another bit of a string, choose to reject its previously hypothesized program even if the previous program has not been falsified. That is, it can happen that $B(S) = z$ and there exists x such that $U(z, x) \geq S \cdot 1$ and nevertheless $B(S \cdot 1) \neq z$. This occurs when the hypothesized program z is no longer economical as part of a description. See Example 1 for $n = 172$.

Bounds on computational power are bounds on the ability to detect structure. The parameter g is suggestive of a measure of intelligence; the greater is g, the greater is the capacity of the learner to detect subtle structure and predict more precisely.

A more general version of non-deterministic bit-generator considered by Levin (see Solomonoff (1978)) is that for which f assigns to strings probabilities of the next bit being, say, 1 without restricting these probabilities to the set {0, 1, 1/2} as we have. It appears clear that the kind of learning-in-the-limit discussed here is not possible for that general case due to the failure of the strong convergence theorem necessary for such learning. However, we leave the reader with the following open problem:

Problem: Is learning possible where the definition of NDBG is as in this article but the native's coin is biased (and the bias is known to the learner)?

References

Bennett, C. (1988). Logical depth and physical complexity. In R. Herken (Ed.), *The universal Turing machine: A half-century survey.* Oxford University Press.

Blum, M., & Blum, L. (1975). Towards a mathematical theory of inductive inference. *Inf. Cont.*, *28*, 125–153.

Chaitin, G.J. (1975). A theory of program size formally identical to information theory. *JACM*, *22*, 329–340.

Dietterich, T., & Michalski, R. (1986). Learning to predict sequences. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*, Los Altos, CA: Morgan Kaufmann.

Gold, E.M. (1967). Language identification in the limit. *Inf. Cont.*, *10*, 447–474.

Kolmogoroff, A.N. (1965). Three approaches to the quantitative definition of information. *Problems of Informati. Transmission*, *1*, 1–7.

Koppel, M., & Atlan, H. (in press). An almost machine-independent theory of program-length complexity, sophistication, and induction. *Information Sciences.*

Levin, L.A. (1973). On the notion of a random sequence. *Soviet Math. Dokl.*, *14*, 1413–1416.

Osherson, D., Stob, M., & Weinstein, S. (1985). *Systems that learn.* Cambridge, MA: MIT Press.

Schnorr, C.P. (1973). Process complexity and effective random tests. *J. Comp. Syst. Sci.*, *7*, 376–384.

Solomonoff, R.J. (1964). A formal theory of inductive inference. *Inf. Cont.*, *7*, 1–22.

Solomonoff, R.J. (1978). Complexity-based inductive systems. *IEEE Trans. on Inf. Th.*, *24*, 422–432.

Wexler, K. & Culicover, P. (1985). *Formal principles of language acquisition.* Cambridge, MA: MIT Press.