Optimization Is Easy and Learning Is Hard In the Typical Function

Thomas M. English
The Tom English Project
2401 45th Street #30
Lubbock, Texas 79412 USA
Tom.English@ieee.com

Abstract-Elementary results in algorithmic information theory are invoked to show that almost all finite functions are highly random. That is, the shortest program generating a given function description is rarely much shorter than the description. It is also shown that the length of a program for learning or optimization poses a bound on the algorithmic information it supplies about any description. For highly random descriptions, success in guessing values is essentially accidental, but learning accuracy can be high in some cases if the program is long. Optimizers, on the other hand, are graded according to the goodness of values in partial functions they sample. In a highly random function, good values are as common and evenly dispersed as bad values, and random sampling of points is very efficient.

1 Introduction

Loosely speaking, the idea of conservation in analysis of learning [1] and optimization algorithms [2] has been that good performance on some problem instances is offset by bad performance on others. For a given random distribution on a problem class and a given performance measure, the condition of conservation is that all algorithms solving that class of problem have identical performance distributions. It has been shown that conservation of statistical information underlies conservation of optimizer performance [3].

The present work diverges from past by addressing conservation in terms of algorithmic information [4]. The information of a problem instance is no longer the extrinsic "surprise" at seeing it as the realization of a random variable, but the intrinsic *complexity* of computing its description. This shift in analytic paradigm makes it possible to characterize what optimizers and learners do on most or all instances of a problem, rather than to characterize the performance distribution. Remarkably, almost every instance exhibits a high degree of algorithmic randomness, and thus has very little internal structure exploitable by programs. Thus conservation is not so much an artifact of the distribution of instances as a consequence of the pervasiveness of algorithmically random instances.

It is shown that an optimizer or learner can essentially reduce the complexity of a particular instance, random or not, by "matching" it. The degree of match, or mutual complexity, is bounded by the length of the program. The essence of conservation of algorithmic information is that a program for exploration or learning cannot reduce the algorithmic complexity of a problem instance by more than its own complexity.

Here optimization and active category learning are given a unified treatment in terms of function exploration (suggested in [3]). The analysis is straightforward, but the unfamiliarity of key concepts will pose problems for some readers. Thus section 2 gives an informal overview of the main results of the paper. Section 3 gives a string representation of functions, describes both variants of function exploration, and formally introduces algorithmic information theory. Section 4 briefly derives conservation of statistical performance, and lays a crucial foundation for section 5 by establishing that function exploration entails implicit permutation of the string representation of the given function. Section 5 derives the main results in conservation of algorithmic information. Sections 6 and 7 give discussion and conclusions.

2 Overview

This section gives an informal introduction to topics that actually require formal treatment. Everything presented here should be taken on a provisional basis.

2.1 Function Exploration

The notion of an optimizer that explores a function to locate points with good values is familiar. The notion of a learner that explores a function is less common. If every domain point belongs to exactly one category, then the association of points with their categories is a function. A learner guesses unobserved parts of the category function on the basis of observed parts. The learner is *active* when it decides on the basis of observations which point's category to guess next. Thus optimizers and active learners both explore functions.

An optimizer is evaluated according to the sequence of values it observes while exploring the function. An active

0-7803-6375-2/00/$10.00 ©2000 IEEE.

learner, on the other hand, is evaluated according to its accuracy in guessing what it will observe. The most straightforward measure of accuracy is the number of correct guesses.

For the sake of analysis, finite functions are represented as binary strings. Each string is a function *description*. The distinction between functions and descriptions is important because the performance of an algorithm on a particular function is generally sensitive to the representation.

The exploration algorithm obtains the value of a point by reading the value (a field of bits) from the appropriate location in the string. The algorithm is required to read all values in finite time, and to write a non-redundant *trace* of the sequence of values it reads. When the algorithm halts, the trace is a permutation of the values in the description. Thus there is no formal distinction between descriptions and traces. A key fact (sec. 4.1) is that the input-output relation of each exploration algorithm is a 1-to-1 correspondence on function descriptions.

2.2 Algorithmic Randomness

The *algorithmic complexity* of a binary string is the length of the shortest program that generates it and halts. The program is loosely analogous to the self-extracting archives commonly transmitted over the Internet. The complexity of the data is essentially the length of the executable archive. Algorithmic complexity is uncomputable, however.

When a string has complexity greater than or equal to its length, it is said to be *algorithmically incompressible*. An incompressible string is also *algorithmically random*. Algorithmic randomness entails all computable tests of randomness [5], and here the term is abbreviated to *random*. If a long string can be algorithmically compressed by at most a small fraction of its length, it may not satisfy the crisp definition of algorithmic randomness, but it is nonetheless *highly random*.

How many strings of a given length are highly random? If the length is great, then almost all of them are. The fraction of strings that are algorithmically compressible by more than k bits is less than 2^{-k} [5]. To appreciate this result, consider the modest example of functions on set of 32-bit integers. Each of 2^{32} domain elements has a 32-bit value, so a function is described by $N = 32 \times 2^{32} = 2^{37}$ bits. Compression by more than $1 / 2^{10} \approx 0.1\%$ corresponds to $k = N / 1024 = 2^{27}$, giving $2^{-k} = 2^{-134217728}$. Compressible descriptions, though plentiful in sheer number, are relatively quite rare.

2.3 Exploration and Complexity

Function exploration has been formulated in such a way as to facilitate reasoning about the algorithmic complexity of the function as described and the function as processed. This line of reasoning does not lead to simple statements about performance, but it does help to characterize the information processing of exploration programs.

The 1-to-1 correspondence of descriptions and traces of a program implies that the mean difference in complexity of description and corresponding trace is zero. The range of differences can vary greatly from one program to another, however. An exploration program cannot generate more algorithmic information than it contains. Suppose that program p explores description x and writes trace y. The difference in complexity of x and y is bounded approximately by the complexity of p. If the difference were to exceed the complexity of p by more than a small amount, then p and the shortest program generating y could be combined with a small amount of code to generate x:

```
Foreach description x Loop
   Execute p with x as input
   If trace matches y Then
      Output x
      Halt
   EndIf
EndLoop
```

This generate-and-test algorithm exploits the invertibility of the mapping from description to trace. The shortest implementation has complexity lower than that of x, but generates x, a contradiction.

This has covered the case of reduction of complexity due to exploration. A program similarly cannot add more than its own complexity to that of the description in generating the trace string. In practice, exploration programs are short in comparison to descriptions, and the complexity ratio of the function as processed and the function as described must be close to unity unless the description is low in complexity.

2.4 Complexity and Performance

There is much about the relationship between complexity and performance that has yet to be investigated. Only simple cases are addressed here, but the results are significant. Performance is assessed differently in learning than in optimization, and it appears that separate treatment is necessary.

2.4.1 Complexity and Optimizer Performance

Assume that the performance criterion is a function of the trace. (In practice, it might be a function of a possibly-redundant record of values observed, rather than a non-redundant record.) For a relatively short optimization program operating upon highly random function descriptions, the traces are also highly random. This implies that a very good value occurs with high probability in a short prefix of the trace. The probability associated with obtaining a certain level of quality in a prefix of a certain length depends only upon the quality, not the size of the function (see sec. 5.3). Remarkably, almost all functions are easy to optimize.

The non-constant functions of lowest complexity are actually the hardest to optimize. These "needle in a haystack" functions assign a good value to exactly one point and a bad value to all the other points. On average, any program must explore half the points to find the good value.

2.4.2 Complexity and Learning Accuracy

The active learning program guesses the trace, rather than the function description. If the program is short relative to a highly random description, the trace is also highly random, and guesses of the trace are correct essentially by chance. That is, about 1 in M guesses will be correct. In this sense, almost all functions are hard to learn.

If a program guesses all values in the trace correctly, this amounts to compression of the function description to a very short length, and the program must be as complex as the description. The construction is omitted here, but the gist is that the learner does not have to write a trace it can guess perfectly, and a contradiction arises if the complexity of the trace is not absorbed into the learner.

3 Formal Preliminaries

3.1 Notation and Conventions

The set of functions under consideration is $F = \{f \mid f\colon S \to \{0, 1\}^L\}$ for indexed $S = \{x_1, \ldots, x_M\}$ and positive L. The *description* of $f \in F$ is the concatenation $f(x_1) \ldots f(x_M) \in \{0, 1\}^N$, where $N = LM$. Every string in $\{0, 1\}^N$ describes exactly one function in F.

Here an optimization or learning algorithm is required to be deterministic, though perhaps pseudorandom with a constant seed, and may be sequential or parallel. Implemented algorithms are referred to as *programs*. As previously indicated, optimization and learning are grouped under the rubric of function *exploration*. An exploration algorithm reads values from function descriptions, and writes binary *trace* strings as outputs (sec. 4.1 gives details). Attention is restricted to exploration algorithms that read every input before halting. To read the value of domain point x_i is to read the i-th field of L bits in the description.

The performance of an optimizer on a given description is a function of the trace. It is assumed that there is some mapping from values in $\{0, 1\}^L$ to goodness values. General characterization of learning accuracy is not so simple. In the present work, the elements of $\{0, 1\}^L$ are taken as category labels, and the learner guesses labels immediately before reading them from the description. The performance criterion is the fraction of correct guesses.

3.2 Algorithmic Complexity

Algorithmic information theory [4] defines the algorithmic complexity of binary strings in terms of halting programs for a universal computer (i.e., an abstract model as powerful as any known). The programs are themselves binary strings, and are required to be self-delimiting. The choice of universal computer is insignificant for large programs, because any universal computer may simulate any other with a program of constant length.

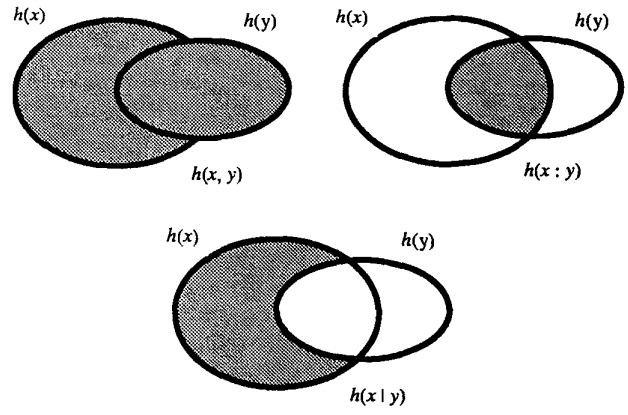

Fig. 1. Complexity is indicated by the size of a region. Thus $h(x) > h(y)$. The shaded regions depict the joint complexity of x and y, $h(x, y)$; the mutual complexity of x and y, $h(x : y)$; and the complexity of x relative to y, $h(x \mid y)$.

Let x and y be strings in $\{0, 1\}^*$. The *algorithmic complexity* of x, denoted $h(x)$, is the length of the shortest program that generates x as output and halts. The *relative complexity* of x given y, denoted $h(x \mid y)$, is the length of the shortest program that generates x as output, given a program that generates y "for free" [4] (see fig. 1). The algorithmic complexity of the pair (x, y) is

$$h(x, y) = h(x) + h(y \mid x) + O(1). \tag{1}$$

$O(1)$ denotes the set of all functions with magnitude asymptotically dominated by some constant. The mutual complexity of x and y is

$$\begin{aligned} h(x : y) &= h(x) + h(y) - h(x, y) \\ &= h(x) - h(x \mid y) + O(1) \\ &= h(y) - h(y \mid x) + O(1) \end{aligned} \tag{2}$$

These identities are closely analogous to ones in conventional information theory. Indeed the algorithmic complexity of random strings is asymptotically equivalent to Shannon entropy:

$$E\, 1/n\; h(X^n \mid n) \to H(X) \text{ as } n \to \infty, \tag{3}$$

where X^n is a sequence of n i.i.d. random variables distributed as X on $\{0, 1\}$ and $H(X)$ is the Shannon entropy of the distribution of X (see theorem 7.3.1 in [5]). Now if a function is drawn uniformly from F, the N bits in the descrip-

tion are i.i.d. uniform on {0, 1} with one bit of entropy apiece. Setting $n = N$ and $X \sim$ Uniform{0, 1} in (3), and assuming that N is large, the expected complexity of the description is N. How can the average complexity be equal to the actual length of the descriptions? Some length-N strings can be generated only by self-delimiting programs that are greater than N in length, even when the programs are given N.

4 Exploration, Permutation, and Conservation

The following derivation of conservation of performance is conventional in its invocation of properties of the distribution of functions. It is unusual, however, in unifying the treatment of conservation of optimization performance [2, 3] and conservation of learning accuracy [1].

4.1 Exploration Is Equivalent to Honest Permutation

Definition. Let A denote a finite alphabet. An algorithm that permutes input string $x = x_1 \ldots x_n \in A^n$ to generate output $\pi(x) = x_{j_1} \ldots x_{j_n}$ is *honest* if it sets j_k without examining x_{j_k}, $k = 1, \ldots, n$.

Any exploration algorithm can be modified to yield honest permutations. With alphabet $A = \{0, 1\}^L$, function descriptions are elements of A^M. Code may be inserted to immediately write elements of A that are read, ensuring that no input is written more than once to the output, and ordering values read in parallel according to input position. In other words, the trace is an honest permutation of the description.

Theorem 1 (π preserves i.i.d. inputs): Let $X = X_1, \ldots, X_n$ be a sequence of random variables i.i.d. on A. If π: $A^n \rightarrow A^n$ is the input-output relation of an honest permutation algorithm, $\pi(X) \sim X$.

Proof: Inputs $X_1, \ldots, X_n$ are identically distributed, so there is no prior distinction between them. By independence, only x_i supplies information about X_i, $i = 1, \ldots, n$, but an honest permutation algorithm does not read any $X_i = x_i$ before setting output index j_i. Thus the output ordering $j_1, \ldots, j_n$ conveys no information about X, and $\pi(X) \equiv X_{j_1}, \ldots, X_{j_n} \sim X$.

Now drawing a string uniformly from A^n is equivalent to sequentially drawing n elements independently and uniformly from A. Thus a uniform distribution of input strings is preserved in the outputs of an honest permutation algorithm. It follows that if every string in A^n is input to the algorithm exactly once, then every string in A^n occurs exactly once as an output:

Corollary (π is bijective): If π: $A^n \rightarrow A^n$ is the input-output relation of an honest permutation algorithm, π is a 1-to-1 correspondence.

Thus any exploration algorithm induces a 1-to-1 correspondence on function descriptions. If the trace is regarded as the "description as processed," then each algorithm processes each description in response to some input.

4.2 Conservation of Optimizer Performance

Assume that the values associated with all domain points are i.i.d. as X on $\{0, 1\}^L$. By thm. 1, the trace values of any optimizer are also i.i.d. as X, and all optimizers have identical trace distributions. Because the performance measure is a function of the trace, it must also be the case that all optimizers have identical performance distributions. Any superiority an optimizer exhibits on a subset of descriptions is precisely offset by inferiority on the complementary subset. This is one sense in which performance is conserved.

4.3 Conservation of Learning Accuracy

A learner not only has to read values, but to predict them. Performance is some function of the guesses and the trace. It is easiest to exhibit conservation of learning accuracy if the descriptions are uniform on $\{0, 1\}^N$ and the performance criterion is the fraction of correct guesses. Under this condition, the category labels in descriptions are i.i.d. uniform on $\{0, 1\}^L$, and, by thm. 1, so are the labels in the trace. Any guessing strategy gets exactly 1 of 2^L guesses correct, on average. A learner may have superior guessing accuracy on one subset of functions, but must compensate precisely on the complementary subset.

5 Complexity and Function Exploration

In contrast to the preceding section, this section derives results that apply to individual function descriptions, rather than to distributions.

5.1 Almost All Descriptions Are Highly Random

Section 3.2 indicated that under the realistic assumption that function descriptions are long, the average complexity of descriptions relative to their length is their length. Consider that a very short program p can be affixed to any description x to obtain a program px that writes x and halts. The complexity $h(px \mid N)$ of a brute force generator of x is at most slightly greater than N. On the other hand, $h(x \mid N) << N$ for some descriptions x. Given this asymmetry, it must be the case that $h(x \mid N) \geq N$ for more than half of all $x \in \{0, 1\}^N$.

It is furthermore the case that almost all descriptions x have $h(x \mid N)$ very close to N. In general, the fraction of strings x in $\{0, 1\}^n$ such that $h(x \mid n) < n - k$ is less than 1 in 2^k, where $0 \leq k < n$. That is,

$$|\{x \in \{0,1\}^n : h(x \mid n) < n - k\}| / 2^n < 2^{-k}. \quad (4)$$

(See thm. 7.5.1 in [5].) With $n = N$, this characterizes both descriptions and traces, due to their 1-to-1 correspondence. In practice, however, programs often generate only a prefix of the trace before halting. Let n be the length of trace prefixes, with n a constant integer multiple of L. Each prefix occurs in 2^{N-n} complete traces, and thus is generated in response to 2^{N-n} descriptions. Because prefixes occur in equal proportion, inequality (4) applies to them as to full traces.

Now $x^n \in \{0,1\}^n$ is said to be *algorithmically random* when $h(x^n \mid n) \geq n$. This entails all computable tests for randomness [5]. Algorithmic randomness is equivalent to algorithmic incompressibility, and compressibility of a finite string is a matter of degree, not an absolute [4]. Thus it is appropriate to say that a string is nearly random when the compression ratio $h(x^n \mid n) / n$ is slightly less than unity. As illustrated in sec. 2.2, compression by even a small fraction of n is very rare when n is large.

In sum, more than half of all descriptions, traces, and trace prefixes strictly satisfy the algorithmic incompressibility criterion for algorithmic randomness. Many others are nearly incompressible. For perspicuity, binary strings x^n with compression ratios $h(x^n \mid n) / n$ close to unity will be referred to as *highly random*. Note that the algorithmic randomness of a description is an intrinsic property, and does not depend upon an extrinsic random distribution on F.

5.2 Conservation of Algorithmic Information

Consider again an exploration program p generating honest permutations $y = \pi(x)$ of descriptions $x \in \{0,1\}^N$. The fact that π is a bijection immediately implies that algorithmic complexity of descriptions is conserved by the program. That is, $\pi(\{0,1\}^N) = \{0,1\}^N$, and

$$|\{x: h(x \mid N) = n\}| = |\{x: h(\pi(x) \mid N) = n\}| \quad (5)$$

for n = 1, 2, 3, ... The complexity histogram for traces is precisely that for descriptions, and does not depend upon the exploration program.

More significantly, there is conservation in the complexity of any description and its corresponding trace relative to the program. Specifically, $h(x \mid p, N) \approx h(\pi(x) \mid p, N)$, and this implies that the absolute difference in complexity of the description and the trace is bounded by the complexity of the program. Thus algorithmic information is conserved in the sense that an exploration program cannot add or take away more information than is present in itself (see fig. 2). In the following, π is the input-output relation of honest permutation program p, and $y = \pi(x)$ for arbitrary $x \in \{0,1\}^N$.

Lemma 1: Given p, x, and N, the complexity of y is bounded by a constant. That is,

$$h(y \mid p, x, N) = O(1). \quad (6)$$

Proof: Construct program p^* that generates output y by invoking p with N and x as input, and then halts. The invocation of given programs may be accomplished with code of constant length, and thus the shortest p^* is constant in length.

Lemma 2: Given p, y, and N, the complexity of x is bounded by a constant. That is,

$$h(x \mid p, y, N) = O(1). \quad (7)$$

Proof: Construct program p^* that enumerates strings $w \in \{0,1\}^N$. Given N, the enumeration may be performed by a constant-length code. For each w, p^* supplies N and w to p as input, and checks to see if the output of p is y. Because π is invertible, there is exactly one w such that $\pi(w) = y$. Thus p^* generates output of $x \equiv w$ and halts when the output of p is y. The comparison of $\pi(w)$ to y can be accomplished with code of constant length, given N. Enumeration, comparison, and invocations of given programs all can be accomplished with code of constant length, and thus the shortest p^* is constant in length.

Theorem 2 (Preservation of relative complexity): The difference in complexity of x and y relative to p and N is bounded by a constant. That is,

$$h(y \mid p, N) = h(x \mid p, N) + O(1). \quad (8)$$

Proof:

$$h(x, y, p \mid N) = h(x, y, p \mid N)$$
$$h(x \mid p, y, N) + h(y, p \mid N) = h(y \mid x, p, N) + h(x, p \mid N) + O(1)$$
$$h(y, p \mid N) = h(x, p \mid N) + O(1),$$

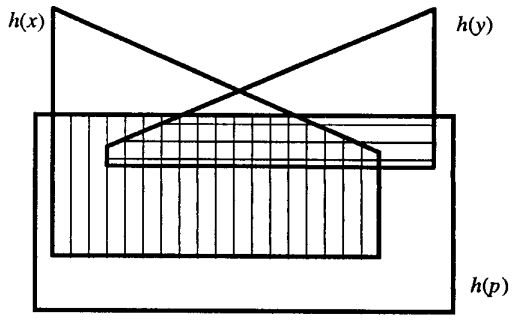

Fig. 2. Conservation of complexity for description x, trace y, and program p. Conditioning upon N is omitted. The x and y regions outside the p rectangle are equal in area because $h(x \mid p) \approx h(y \mid p)$. Thus any difference in x area and y area must be accounted for within the p rectangle. That is, the difference in area of the hatched regions of x-p overlap and y-p overlap is the complexity difference of x and y, and $h(x) - h(y) \approx h(x : p) - h(y : p) \leq h(p)$.

by lemmas 1 and 2. Subtraction of $h(p \mid N)$ from both sides of the relation yields (8).

In the following theorem, note that both positive and negative constants are $O(1)$. Conditioning on N is implicit in all complexity expressions.

Theorem 3 (Conservation): The magnitude of information gain in the output of p is bounded by the complexity of p. That is,

$$\begin{aligned} |h(x) - h(y)| + O(1) &= |h(x : p) - h(y : p)| && \text{(9a)} \\ &\le \max\{h(x : p), h(y : p)\} && \text{(9b)} \\ &\le h(p). && \text{(9c)} \end{aligned}$$

Proof: To establish (9a), take the difference of

$$h(x) = h(x \mid p) + h(x : p) + O(1)$$

and

$$\begin{aligned} h(y) &= h(y \mid p) + h(y : p) + O(1) \\ &= h(x \mid p) + h(y : p) + O(1) \qquad \text{[by Thm. 2].} \end{aligned}$$

Inequality (9b) follows from the non-negativity of $h(x : p)$ and $h(y : p)$. Inequality (9c) holds because $h(w : p) \le h(p)$ for all $w \in \{0, 1\}^N$.

Corollary (Conservation in prefixes): If $y = y'z = \pi(x)$, $y' \in \{0, 1\}^m$, $0 < m < N$, and $z \in \{0, 1\}^n$, then

$$h(y' \mid m) \ge h(x \mid N) - h(p) - l^* - n + O(1), \qquad (10)$$

where $l^* = \log^* \min\{m, n\}$.

Proof: The inequality

$$h(y \mid N) \le l^* + h(y' \mid m) + h(z \mid n) + O(1) \qquad (11)$$

is derived by constructing a program that, given N, invokes programs to write y' (given m) and z (given n) successively. The value of $\min\{m, n\}$ is stored in the program in a self-delimiting form that requires $l^* = \log^* \min\{m, n\}$ bits [5]. The value of $\max\{m, n\}$ is computed as $N - \min\{m, n\}$. The length of the code for subtraction and invocations is $O(1)$. Thus the right-hand side of (11) gives the length of a program that, given N, generates y. Eqn. (10) is obtained from (11) by replacing $h(y \mid N)$ with its minimum value of $h(x \mid N) - h(p)$, replacing $h(z \mid n)$ with its maximum value of $n + O(1)$, and rearranging terms.

Observation: If both y' and z are algorithmically random, l^* can be omitted from (10).

5.3 Optimization Is Almost Always Easy

It was established in sec. 5.1 that the fraction of function descriptions for which trace prefixes of a given length are compressible by more than k bits is less than 2^{-k}. For prefixes containing $m \le N / L$ values, almost all are highly random if m is large. This implies a high degree of dispersion of values over the codomain. For instance, both 0 and 1 must occur in every position of the L-bit values in the prefix, or it is possible to compress the prefix simply by omitting constant bits. If the values are interpreted as unsigned integers, this alone guarantees that a value in the upper half of the codomain occurs in the prefix. But it must also be the case that 0 and 1 are approximately equiprobable in each position of the integers in the prefix, or there is a scheme for compressing the prefix. Thus approximately half of the values are in the better half of the codomain, approximately one-fourth are in the best quartile of the codomain, etc. Thus highly random trace prefixes are quite benign in the context of optimization.

Assuming that the prefix y' comprises less than half of algorithmically random trace y, it can be inferred from (10) that

$$h(y' \mid m') \ge m' - h(p) - \log^* m' + O(1), \qquad (12)$$

where $m' = mL$ is the length in bits of an m-value prefix. To appreciate how $\log^* m'$ is dominated by other terms in (12), consider that $m' = 2^{25}$ gives $\log^* m' \approx 36.2$. Thus any long trace prefix generated by a short program is highly random when the description is algorithmically random.

A nonparametric approach to assessing the difficulty of optimization in almost all cases is to select an arbitrary function description $x \in \{0, 1\}^N$ and derive the fraction of optimizers that achieve a certain level of performance. The infinitude of optimization programs with the same input-output relation is problematic, however, so the fraction of permutations of the description giving a certain level of performance is derived instead. The assumption is that each permutation of x is implemented by as many programs as every other permutation.

What fraction of permutations of x yield a value as good as θ among the first m values? Let $n = N / L$ be the number of values in x, and let k be the number of values in x that are worse that θ. Then the fraction of permutations of x that do *not* contain a value as good as θ in the m-prefix is

$$\frac{k!(n-m)!}{n!(k-m)!} \le \left(\frac{k}{n}\right)^m. \qquad (13)$$

Interestingly, (13) arises in counting functions for which a fixed optimizer fails to obtain the threshold value [6, 7]. For $n = 2^{32}$ points, $k = 0.99999n$, and $m = 10^6$, all but 4.5×10^{-5} of permutations include a one-in-ten-thousand value among the first million values. Thus almost all optimizers discover good values rapidly. It bears mention that this analysis, unlike that at the beginning of the subsection, defines "good" in terms of the range of the described function., and not the codomain.

5.4 Learning Is Almost Always Hard

Learners have the disadvantage, relative to optimizers, of being scored on their *prediction* of all values in the trace, not their *discovery* of good values. For a highly random description, the learner attempts to infer regularity from random values and use that (nonexistent) regularity to predict random values. Learning to guess well is equivalent to reducing the uncertainty of unread values, and that implies data compression. But almost all descriptions are compressible by at most a small amount, and accurate guessing is rare.

Learning performance is evaluated in terms of the entire trace. In practice, the number of distinct category labels, $k = 2^L$, is much less than $n = N / L$, the number of labels in the trace. Under this condition, any highly random trace contains approximately the same number of instances of each label. Although the point will not be argued formally, this gives some insight as to why the typical accuracy, and not just the mean accuracy, of a learning program is 1 in k correct.

Now let y denote any trace. The approach is to hold y constant and determine the fraction of learners achieving a given level of guessing accuracy. To simplify, it is assumed that each sequence of guesses is generated by an identical number of programs, allowing sequences to be counted instead of programs. The fraction of sequences with m correct guesses has the form of the binomial distribution, $b(m; n, p)$, $p = 1 / k$, though it is not random. To determine the fraction of sequences with m or fewer correct guesses, the standard normal approximation $z = (m - np) / (npq)^{1/2}$, $q = 1 - p$, is convenient. To appreciate the rarity of learners that have accuracy as good as 1.001 times the chance rate, let the domain have $n = 2^{32}$ points, let the number of categories be $k = 2^6 + 1$, and let the number of correct guesses be $m = (1 + 2^{-10})\ pn$. The resulting z-value is 8. Thus very few learners get more than 1.001 of k guesses correct.

6 Discussion

6.1 Near Ubiquitity of Highly Random Trace Prefixes

The length of trace prefixes does not have to be great for it to be the case that almost all prefixes are highly random. Exploration of $2^{20} \approx 10^6$ points is common in applications. If an exploration program halts after reading the values of 2^{20} domain points, and each point has a 32-bit value, then the program generates a trace prefix of $n = 2^{25}$ bits. Compression of $n / 1024$ bits gives $k = 2^{15}$ in (4). That is, fewer than 1 in 2^{32768} functions yields a trace prefix compressible by more than 0.1% of its length.

6.2 Conservation of Algorithmic Information

A difficulty in understanding conservation of algorithmic information (sec. 5.2) is that it is inherently backhanded. The crucial point is that an exploration program is equally uninformed of a function description and the corresponding trace, but not equally informed. This constraint of $h(x \mid p, N) \approx h(\pi(x) \mid p, N)$ arises from invertibility of the program's mapping from descriptions to traces. The synopsis in fig. 2 of this result and its ramifications is perhaps the best aid to intuition.

6.3 Highly Random Function Descriptions

It is essential to understand that the results of sections 5.1, 5.3, and 5.4 depend upon the fact that almost all function descriptions are intrinsically random objects, and not upon a random distribution of functions. The uniform distribution is assumed only in determining what fraction of descriptions are highly random. The focus is upon performance of algorithms on almost all descriptions, individually, not their performance distributions for a function distribution. Thus the results may be characterized as distribution-free.

It should be noted, nonetheless, that for almost all descriptions it is impossible to reject with much confidence the hypothesis that the description's bits were generated by flipping a fair coin N times. Thus almost every description provides properties that have been exploited in prior work with the uniform distribution of functions. The condition of values that are independent and uniform on the codomain does not arise merely when all function distributions are averaged to obtain the uniform. It holds approximately for individual functions, with relatively few exceptions. In this sense the uniform is not just an average, but is nearly ubiquitous.

Any distribution on the set of highly random descriptions is a boon to most or all optimizers and the bane of most or all learners. In the extreme case that all probability mass is assigned to one random description, by virtue of high mutual complexity with the description some complex optimizers will generate a trace with values in reverse order of goodness and some complex learners will guess perfectly. But with its dearth of structure, the description inherently does not hide good values from optimizers and does not allow generalization from observed to unobserved values.

7 Conclusion

Elementary algorithmic information theory has facilitated reformulation of conservation of information [3] in terms of individual functions. The issue of the distribution of functions has been circumvented. The notion, previously rather nebulous [3], that a program could excel only by encoding prior knowledge has been made concrete. The algorithmic information supplied by an optimization or learning program is roughly bounded by its own length, whether or not the explored function is highly random.

It has been shown that category learners with high accuracy are extremely rare. In optimization, on the other hand, it has been shown that for the vast majority of functions there is no strategy better than to visit a large number of points as rapidly as possible. Indeed, such a strategy works extremely well, and the epithet of "just random search" has no basis in formal analysis.

Acknowledgment

Thanks to Nicole Weicker for her constructive remarks.

References

[1] Schaffer, C. 1994. "A conservation law for generalization performance," in *Proc. Eleventh Int'l Conf. on Machine Learning*, H. Willian and W. Cohen, eds. San Francisco: Morgan Kaufmann, pp. 259-265.

[2] Wolpert, D. H., and W. G. Macready. 1997. "No free lunch theorems for optimization," *IEEE Trans. Evolutionary Computation*, vol. 1, no. 1, pp. 67-82.

[3] English, T. M. 1999. "Some Information Theoretic Results on Evolutionary Optimization," in *Proc. Of the 1999 Congress on Evolutionary Computation: CEC99*. Piscataway, New Jersey: IEEE Service Center, pp. 788-795.

[4] Chaitin, G. J. 1999. *The Unknowable*, New York: Springer-Verlag, chap. 6.

[5] Cover, T. M., and J. A. Thomas. 1991. *Elements of Information Theory*, New York: Wiley & Sons.

[6] Breeden, J. L. 1994. "An EP/GA synthesis for optimal state space representations," in *Proc. Third Annual Conf. on Evolutionary Programming*, A. Sebald and L. Fogel, eds. River Edge, NF: World Scientific, pp. 216-223.

[7] English, T. M. 1996. "Evaluation of evolutionary and genetic optimizers: No free lunch," in *Evolutionary Programming V: Proc. Fifth Ann. Conf. on Evolutionary Programming*, L. Fogel, P. Angeline, and T. Bäck, eds. Cambridge, Mass.: MIT Press, pp. 163-169.