Some Notes on Kolmogorov Complexity, the Algorithmic Markov Condition, and Causality

David Meyer

dmm@{1-4-5.net,uoregon.edu,...}

Last update: August 16, 2018

1 Introduction

The Kolmogorov complexity of a string s, $K(x)$, is generally be defined as a function from finite binary strings of arbitrary length to the natural numbers $\mathbb{N}$ [1]. That is, $K : \{0,1\}^* \to \mathbb{N}$ is a function defined on *objects* represented by binary strings. As we shall see, this definition is can be extended to other types of objects such as numbers, sets, functions and probability distributions.

As a first approximation, $K(x)$ may be thought of as the length of the shortest computer program that prints x and then halts. This computer program may be written in C, Java, Python or any other universal programming language. These are programming languages in which a universal Turing Machine can be implemented. Many if not most programming languages have this property. Here we will just fix some universal language (say, python) and define Kolmogorov complexity with respect to it. The invariance theorem (Section 2.3) tells us that it does not really matter which language we pick.

Note that computer programs often make use of data, and frequently such data are encoded inside the program itself. An example is the bitstring "010110..." in the program

$$\text{print "}\underbrace{010101\ldots01}_{\text{n times "01"}}\text{"} \tag{1}$$

So what is the complexity of the program shown in 1? We think of the "interpreter" for this program as a *Universal Machine* (in particular, a Universal Turing Machine or UTM), so all programs have a constant (call it C) complexity that accounts for the UTM. What is the Kolmogorov complexity of such a program? Well, since a program to print this string and halt, call it p, might be (in this case, python):

print "01" * n

Here we notice that we need only $O(\log n)$ bits to represent the number n, the number of times the string "01" is repeated. So for this program $K(p) = O(\log n)$. Other programs that have $K(p) = O(\log n)$ include printing out the first n digits of π (or e). So while the decimal expansion of π might appear to be random, it has low Kolmogorov complexity (in fact, $O(\log n)$) since the digits of π can be generated by a simple program where the only quantity that is a function of n is n itself, and we need $O(\log n)$ bits to represent n.

A bit of notation: We use x to denote finite bitstrings $x \in \mathbb{B}^*$. Also, let $l(x)$, the length of a given bitstring x, be n. That is, $n := l(x)$. Boldface $\mathbf{x}$ denotes a (possibly infinite) binary string and $x_{[1:n]}$ is the n-bit prefix of $\mathbf{x}$.

2 Properties of $K(x)$

2.1 Very Simple Programs

For very simple objects, $K(x) = O(\log n)$. Here $K(x)$ must be small for 'simple' or 'regular' objects x. For example, there exists a fixed-size program that, when input n, outputs the first n bits of π and then halts (see above). It is easy to see that this specification of n takes $O(\log n)$ bits. Thus, as we saw above, when x consists of the first n bits of π, i$K(x) = O(\log n)$.

Note values of n, $K(x)$ may be quite a bit smaller than $O(\log n)$. For example, suppose $n = 2^m$ for some $m \in \mathbb{N}$, or said differently, $m = \log n$. Then we can describe n by first describing m. To do so we need to describe a program implementing the function $f(z) = 2^z$. Note that the description of the program that computes $f(z)$ takes a constant number of bits which does not depend on n. Now, we know that the description of m takes $O(\log m)$ bits, and we also know that $m = \log n$, so if n has this form we get $K(x) = O(\log m) = O(\log \log n)$.

2.2 Completely Random Objects

Here $K(x) = n + O(\log n)$. Here we can think of a code or description method as a binary relation between source words, namely, strings to be encoded and code words (the encoded versions of these strings). Without loss of generality, we can take the set of code words to be finite binary strings [2]. In these notes we'll only consider uniquely decodable codes where the relation is one-to-one or one-to- many[1], indicating that given an encoding $E(x)$ of string x, we can always reconstruct the original x. The Kolmogorov complexity of x, $K(x)$, can be viewed as the code length of x that results from using the Kolmogorov code $E^*(x)$: this is the code that encodes x by the shortest

[1] We'll need this 1:1 or many:1 property counting arguments.

program that prints x and halts. Another way to think about this: if $K(x) \geq n$ then x is "Kolmogorov random"; we in this case we might also refer to x as "incompressible". See also [3].

One of the key insights relating to Kolmogorov codes is as follows: for any uniquely decodable code, there are no more than 2^n strings x which can be described by n bits. The reason for this is that, as we know, there are no more than 2^n binary strings of length n. However, the number of strings that can be described by less than n bits, that is, $2^0 + 2^1 + 2^2 + \ldots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i = 2^n - 1$, and $2^n - 1 < 2^n$; thus by the Pigeon Hole Principle there is at least one string of length n which can not be mapped to a shorter string[2].

We can use the same argument to observe that the fraction of strings x of length n with $K(x) < n-k$ is less than 2^{-k}. Why? We know that $2^0 + 2^1 + 2^2 + \ldots + 2^{n-1} < 2^n$. Now, suppose that the 2^n strings are uniformly distributed (that is, all strings have equal probability, namely, 2^{-n}). Then there are 2^n strings of length n, and $2^{(n-k)}/2^n = 2^{(n-k-n)} = 2^{-k}$. So no more than 2^{-k} of the 2^n strings are compressible. On the other hand, most of the strings of length n, notably the fraction $1 - 2^{-k}$, aren't compressible by more than a constant. Somewhat surprising. Said another way, we know that $K(x) \geq n - k$ with probability at least $1 - 2^{-k}$.

To avoid variability due to encoding, we encode x in a prefix-free manner (see Section 2.4 on prefix codes), which requires $n + O(\log n)$ bits, rather than $n + O(1)$. Therefore, for all x of length n, $K(x) \leq n + O(\log n)$. Now, as we saw above, except for a fraction of 2^{-k} of these $K(x) \geq n - k$, so we see that for the overwhelming majority of x, $K(x) = n + O(\log n)$.

2.3 Invariance

It would seem that $K(x)$ depends strongly on what programming language we used in our definition of K. However, it turns out that for any two universal languages L_1 and L_2

$$|K_1(x) - K_2(x)| \leq C \tag{2}$$

where K_1 and K_2 denote the respective complexities of L_1 and L_2 and C is a constant that depends on L_1 and L_2 but not on x or its length. So any two universal machines differ by only a constant.

2.4 Prefix Codes

Let's start with a bit of notation: Let $\mathcal{X}$ be some finite or countable set[3]. As usual, the notation $\mathcal{X}^*$ (or $\mathbb{B}^*$) is used to denote the set of finite strings or sequences over $\mathcal{X}$. For example, we may take $\mathcal{X}^* = \{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$, where ϵ represents the empty string, that is,

[2]Such a string has $K(X) \geq |x|$, that is, x is not compressible.

[3]Sometimes the symbol $\mathbb{B}$, for "binary" I assume, is used instead of $\mathcal{X}$.

the string with no characters. The length $l(x)$ of x is the number of bits in the binary string x. For example, $l(010) = 3$ and $l(\epsilon) = 0$. If x is interpreted as an integer, we get $l(x) = \lfloor log(x + 1) \rfloor$, and for $x \geq 2$,

$$\lfloor \log x \rfloor \leq l(x) \geq \lceil \log x \rceil \tag{3}$$

where $\lceil x \rceil$ is the smallest integer larger than or equal to x and $\lfloor x \rfloor$ is the largest integer smaller than or equal to x, and log denotes logarithm to base two. Kolmogorov theory is typically be concerned with encoding finite-length binary strings by other finite-length binary strings; binary strings are only used as a convenience since observations in any alphabet can be so encoded in binary.

One of the problems we face with many codes is that in general, we cannot uniquely recover x and y from their concatenated encoding $E(xy)$. To see this, let E be the identity mapping ($E(X) = x$). Then $E(11)E(11) = 1111 = E(111)E(1)$. As we will see below, prefix-free codes are one way to address this problem. First, a bit of notation.

A binary string x is a proper prefix of a binary string y if we can write $y = xz$ for $z \neq \epsilon$. A set $\{x, y, \ldots\} \subseteq \{0, 1\}^*$ is prefix-free if for any pair of distinct elements in the set neither is a proper prefix of the other. A function $D : \{0, 1\}^* \to N$ defines a prefix-free code if its domain is prefix-free. Decoding a prefix-free code is relatively simple: start at the beginning and decode one code word at a time. When we come to the end of a code word, we know it is the end, since no code word is the prefix of any other code word in a prefix-free code. Clearly, prefix-free codes are uniquely decodable: we can always unambiguously reconstruct an outcome from its encoding. Note that prefix codes are not the only codes with this property; there are uniquely decodable codes which are not prefix-free. In the next section, we will define Kolmogorov complexity in terms of prefix-free codes.

2.5 Prefix-free Codes for $\mathbb{N}$

Suppose we encode each binary string $x = x_1 x_2 \ldots x_n$ as

$$\bar{x} = \underbrace{1111 \ldots 1}_{\text{n times}} 0 x_1 x_2 \ldots x_n \tag{4}$$

We can see that $\bar{x}$ is prefix-free: we can determine where the code word $\bar{x}$ ends by reading it from left to right without the need to back up. $\bar{x}$ has some interesting properties. First, note $l(\bar{x}) = 2n+1$; here we have encoded strings in $\mathbb{B}^*$ in a relatively simple prefix-free manner; the cost is doubling the length of the encoding. There are, however, approaches that give us a much more efficient code. One such approach is to apply the $\bar{x}$ encoding (Equation 4) to $l(x)$ rather than to x. To do so define $x' = \overline{l(x)}x$, where $l(x)$ is interpreted as a binary string of length $\log l(x)$, where $\log l(x) = \log n$. Then the code that maps x to x' is a prefix-free code satisfying $\forall x \in \{0, 1\}^*,\ l(x') = n + 2 \log n + 1$

(ignoring rounding errors). This is because the length of the prefix-free code described in Equation 4 is $2n + 1$, here n represents the length of $l(x)$, which is $\log n$. So we have the length of $\overline{l(x)} = 2\log n + 1$ so that $l(x') = l(\overline{l(x)}x) = 2\log n + 1 + |x| = n + 2\log n + 1$. This code is frequently called the *standard prefix-free code for the natural numbers* and $L_{\mathbb{N}}(x)$ represents the code length of x under this code. That is, $L_{\mathbb{N}}(x) = l(x') = n + 2\log n + 1$.

References

[1] D. Janzing and B. Schölkopf, "Causal inference using the algorithmic markov condition," *IEEE Trans. Inf. Theor.*, vol. 56, pp. 5168–5194, Oct. 2010.

[2] T. M. Cover and J. A. Thomas, *Elements of Information Theory 2nd Edition (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, July 2006.

[3] D. Janzing and B. Steudel, "Justifying additive-noise-model based causal discovery via algorithmic information theory," *ArXiv e-prints*, Oct. 2009.