Course notes for Data Compression - 2
Kolmogorov complexity

Fall 2005

Peter Bro Miltersen

September 29, 2005

Version 2.0

1 Kolmogorov Complexity

In this section, we present the concept of Kolmogorov Complexity that will allow us to escape the paradox of data compression we presented previously. Recall the "Encyclopedia Britannica code" that compresses the Encyclopedia Britanica to one bit. So far, our answer to the paradox has been to only consider optimality of compression relative to a data model, i.e., to postulate some uncertainty about the data we want to compress. While this gives a very elegant theory, it does not give us any answer to how much an individual file, such as the Encyclopedia Britannica may "really" be compressed in absolute terms.

It is interesting to compare $x =$ Encylopedia Britanica with another string of approximately the same length: Let y be string consisting of the first 10 million decimal digits of the number π. If we consider compressing y vs compressing x from the point of view of statistical coding theory, nothing suggests that y would be much more compressible than x. In fact, any of the standard models we have looked at would assign to y a self-entropy roughly equal to the entropy of 10 million digits chosen independently and uniformly at random from $\{0, 1, 2, \ldots, 9\}$. Yet intuitively, y seems much more compressible than x, since the string "the first 10 million decimal digits of π" in a sense contains all the information that y does.

We can explain the difference between x and y by pinpointing what it is that bothers us with the Encyclopedia Britannica codec that compresses x to 1 bit: *The decoder (and coder) must have the Encyclopedia Britannica built in.* Thus, if we must include the size of the decoder in our estimate of the code length, we can no longer compress x to one bit. On the other hand, the string y has a short representation even if we must include the size of the decoder, if we let the decoder be a program that computes a given number of digits of π.

To formalize the above considerations, we need to fix a programming language in which to specify the decoder. We do this by the notion of a programming system.

A *programming system* is a language $L \subseteq \{0, 1\}^*$. We demand that no string in L is a prefix of another string in L. Each string in L is called a *program*. There should be an effective algorithm deciding whether a given string p is a valid program or not, i.e., whether p is in L or not.

The reader may think of a standard programming language such as java as a programming system. The fact that we demand programs to be binary strings is easily handled using a standard fixed length code for any desired

bigger alphabet, and the fact that no string in L is a prefix of another string is true for most well structured programming languages anyway and if not, it is easy to modify the language slightly so that it is true.

To fix the semantics of the programming languauge we need to specify how the programs take inputs and produce outputs. For reasons that will become clear, we want a program $p \in L$ to take as input a (potentially infinite) *stream* of input bits. If p halts, it can have read only a finite prefix of the stream and this prefix is well-defined. Also, if p is given a different stream as input starting with the same prefix, it will behave in exactly the same way. As our program p is intended to model a decoder, it should output a string over a finite alphabet if it halts[1]. Finally, we want our system to be Turing equivalent, i.e., to have the same power as Turing machines.

Note that a natural Turing machine model directly fullfills our input/output format requirements; we simply consider Turing Machines with an arbitrary number of work tapes and two special tapes: an input tape and an output tape. The input tape is semi-infinite and the head on the input tape is read-only and move-right-only. Each cell of the tape contains a bit, 0 or 1. The ouput tape is also semi-infinite and the head on the output tape is write-only and move-right-only. There are no restrictions on the other tapes. When a machine of this kind runs and halts, it will have read a well-defined finite prefix of its input tape and produced some well-defined string on its ouput tape.

To formalize the definition of the semantics of a programming system, we directly relate it to the Turing machine model just specified:

A programming system L must have two associated algorithms ("cross-compilers") ϕ, τ so that $\phi(p)$ for $p \in L$ outputs the index of a Turing machine $M_{\phi(p)}$ of the above kind. The input/output behavior of this Turing machine is the semantics of the program p. Formally, we define

$$p(x) := M_{\phi(p)}(x),$$

where x is an infinite $\{0,1\}$-string and $M_{\phi(p)}(x)$ is either

- $\perp$, when $M_{\phi(p)}$ fails to halt when given x on its input tape, or
- a string y which is the string $M_{\phi(p)}$ has written on its output tape when it halts on input x.

[1]To avoid serious problems concerning the specification of *which* alphabet, a "finite alphabet" means, in this section, the set $\{0, 1, \ldots, s-1\}$ for some positive integer s

The *legal inputs* $I(p)$ of p is the set of finite $\{0,1\}$-strings that are the set of tape prefixes actually having been read by p when halting on some input. Formally,

$$I(p) := \{x| \text{ On input } xz \text{ where } z \in \{0,1\}^\infty, M_{\phi(p)} \text{ reads exactly } x \text{ and halts.}\}$$

Note that by definition, no string in $I(p)$ is the prefix of another string in $I(p)$. For $x \in I(p)$ we define

$$p(x) := p(xz)$$

where z is an arbitrary infinite $\{0,1\}$-string. For $x \notin I(p)$, we let $p(x) = \perp$.

The algorithm τ performs "cross-compilation" in the opposite direction from ϕ. The existense of τ ensures that our programming system is Turing-complete. The algorithm τ maps indices of Turing machines to programs and must satisfy that $\tau(i) = p \Rightarrow p(x) = M_i(x)$ for all indices i and all infinite $\{0,1\}$-strings x. This completes the definition of a programming system.

We are now ready for our main definition. The *Kolmogorov decoder* relative to the programming system L is the partial map

$$d_K(pz) := p(z).$$

Note that since no two strings of L are prefixes of one another d_K is a well-defined map on $\{0,1\}^*$. Since $I(p)$ has the same prefix-freeness property, the the set of strings y in $\{0,1\}^*$ for which $d_K(y) \neq \perp$ also has the prefix-freeness property. Finally note that d_K is computable by the existence of the map ϕ and the existence of a universal Turing machine (with a non-terminating computation corresponding to the value $\perp$).

We think of px as a code word representing $p(x)$. Note however, that any particular string has infinitely many code words representing it. The code word of interest is the shortest one. Thus, we define the *Kolmogorov code*:

$$c_K(x) := \text{ the shortest } y \text{ so that } d_K(y) = x$$

where ties are broken in an arbitrary way (or not broken at all, i.e., we may allow c_K to be multiple-valued).

Finally, the length of the Kolmogorov code for a string x is called the *Kolmogorov complexity* $K(x)$ of x, i.e.,

$$K(x) := |c_K(x)|.$$

Intuitively, the Kolmogorov complexity of x is the absolute limit of the size of a compressed representation of a given string if the size of the decoding program is included in the size bound.

It is easy to see that our choice of programming system only matters up to an additive constant:

Proposition 1 *If L and L' are two different programming systems and K and K' the corresponding notions of Kolmogorov complexity, then there are constants α, β so that for all x, $K'(x) \in [K(x) - \alpha; K(x) + \beta]$.*

Here α is the size of an interpreter for L' written in the language L and β is the size of an interpreter for L, written in the language L'.

Since the Kolmogorov code is a prefix code we have by Shannon's theorem that the average value of $K(x)$ among all strings in $\{0,1\}^n$ is at least n, i.e., on the average, strings are incompressible. By the program reading the Elias encoding of a number n followed by n bits which are copied to the output tape, we also have for any $x \in \{0,1\}^n$ the upper bound $K(x) \leq n + O(\log n)$, i.e., a typical string of length n has Kolmogorov complexity around n. A string of length n of Kolmogorov complexity at least n is called an *algorithmically random* string. An algorithmically random string cannot be compressed if the length of the decoder is taken into account. However, if there is any way of compressing a string using a decoder described by a short program the Kolmogorov complexity can be much smaller than n.

Let us return to our comparision between the strings $x =$ the Encyclopedia Britannica and $y =$ the first 10,000,000 million digits of π. In general, if we let π_n be the first n digits of π, we have that there is a constant c so that $K(\pi_n) \leq c + \ell(n)$ where $\ell(n)$ is the length of the Elias code for n and c is the length of a program that outputs the first n digits of π on input $c_{\text{Elias}}(n)$. This program can be made quite small in natural programming languauges, so relative to these the Kolmogorov complexity will be quite small. We expect the Kolmogorov complexity of the Encyclopedia Britannica to be bigger, but the Encyclopedia is clearly not an algorithmically random string. In particular, we can compress it using the method of statistical coding and some suitable model. Hence, the following proposition becomes relevant.

Proposition 2 *For any programming system, there is a constant c so that the following holds. Let M be a prediction model given by a program p_M, including a specification of the message length. Then for any string x,*

$$K(x) \leq H_M(x) + |p_M| + c.$$

Here, $H_M(x)$ is the self-entropy of x within the model M. The proposition follows from the results we proved about arithmetic coding: The constant c is essentially the length of a program implementing arithmetic decoding, taking as inputs a program specifying the prediction model and the artihmetic code word. Thus, the proposition nicely captures the fact that an arithmetic code has as overhead the specification of the model used to do the compression.

The importance of the concept of Kolmogorov complexity for practical data compression lies mainly in its putting a rigourous meaning to the intuitive idea that there is a specific limit beyond which compression of an individual message is impossible, thus resolving the paradox from the beginning of these notes. Just having in mind that the Kolmogorov complexity of any file is a well-defined number can be useful. Using the notion, we can put forward formally meaningful hypotheses stating that particular files cannot be compressed beyond certain bounds using *any* means, rather than restricting such hypotheses to compression using certain methods. Such a hypothesis can be falsified by actually coming up with a program and input pair beating the hypothesized bound. However, we shall see in the next session that it is in general impossible to *verify* such hypotheses mathematically and thus, even though they are mathematically well-defined, we have to treat them similarly to *scientific* hypotheses which have the same property according to the theory of science: By the scientific method they can be falsified but not verified.

2 Uncomputability of the Kolmogorov code

As a consequence of the following theorem, the Kolmogorov code $x \to c_K(x)$ is uncomputable.

Theorem 3 *The map $x \to K(x)$, $x \in \{0,1\}^*$ is uncomputable.*

Proof Suppose K can be computed by some program. Then consider the program in Figure 1. We know that for every n, there is some binary string x of length n so that $K(x) \geq n$. Hence, for any Elias-coded input n, the program will output a string of length n and Kolmogorov complexity at least n and halt. But the program has some representation p in our programming system, so a code word for its output on input n in the Kolmogorov code would be $pc_{\text{Elias}}(n)$. But $|pc_{\text{Elias}}(n)| = O(\log n)$, so for sufficiently large n, we get a contradiction.

There are some interesting things to notice about the proof.

input EliasCode(n)
for $x \in \{0,1\}^n$ **do**
  if $K(x) \geq n$ **then output** x; **halt fi**
od

Figure 1: Program that would exist if K was computable.

input EliasCode(n)
Simulate method A until a statement "$K(x) \geq i$" for some $i \geq n$ is produced.
output x.

Figure 2: A program that cannot exist.

- It is an uncomputability proof that is *not* a reduction from the undecidability of the halting problem H. It *is* possible to show the uncomputability of K by such a reduction (K and H are in fact Turing-equivalent), but such a proofs are more gritty.
- It is arguably a bit simpler than the standard proof of the undecidability of the halting problem. We don't need the somewhat mind-boggling idea of giving a program itself as an input in a direct way.

That the Kolmogorov code is uncomputable just means that we cannot compute it on all inputs. May we compute it on many? We know that the halting problem is easy on infinitely many (natural) instances and can by automatically decided using verification methods on those. In contrast, we can show the following theorem that states that any correct method yielding statements "$K(x) \geq i$" for strings x and numbers i can yield such statements for only finitely many i.

Theorem 4 *Let an algorithmic method A be given that produces (without input) a stream of correct statements "$K(x) \geq i$". Then, there is a constant c depending on the method, so that $i \leq c$ for all statements "$K(x) \geq i$" produced by the method.*

Proof Suppose not. Then the method A produces statements "$K(x) \geq i$" for arbitrarily large i. Then consider the program in Figure 2. The same way as in the previous proof, we get a contradiction: The program is guaranteed to produce a string of Kolmogorov complexity at least n, yet our analysis shows that its Kolmogorov complexity is $O(\log n)$.

Corollary 5 *Let an algorithmic method A be given that produces (without input) a stream of correct statements "$K(x) = i$". Then, the method only produces finitely many different statements.*

Proof By theorem 4, there is a constant c, so that only statements "$K(x) = i$" with $i \leq c$ are produced. But at most 2^c different x's have $K(x) \leq c$.

In summary, the notion of Kolmogorov complexity allows us to use specific fixed files (such as the Canterbury Corpus) as benchmarks for data compression without having to worry about running into a cheating "Encyclopedia Britannica code". For any such benchmark file, we can keep track of the current compression record in a meaningful way by fixing a programming system and insisting that the size of the decoder is taking into account (For real world systems such as Microsoft Windows, we could insist that the compressed file is a "self-extracting archive"). But by Corollary 5, we will not be able to prove that any given record will never be broken, no matter which method we try to use to prove such an optimality result.

A natural example of a "method" to which Theorem 4 applies is generating true statements from a particular set of logical axioms and using particular rules of inference. Thus, we may take a logical system powerful enough to encode all of mathematics such as set theory, i.e., ZFC. The system can also express statements "$K(x) \geq i$" or more precisely, for each statement "$K(x) \geq i$" there is a corresponding well-defined ZFC statement that can be algorithmically recognized as the equivalent of "$K(x) \geq i$". We can therefore take ZFC and make a program that generates an infinite stream of all theorems provable in ZFC and outputs those that correspond to statements "$K(x) \geq i$". Theorem 4 now implies that (unless ZFC is inconsistent), we never get a statement of the form "$K(x) \geq c$" where c is, essentially, the size of our dovetailing program. In other words, ZFC can only prove a statement of the form "$K(x) \geq i$" for finitely many i. More concretely, if we fix the programming system to an idealized version of Java with only the most necessary built-in classes, the constant c is surely (much) less than 3 Mb. Thus we have shown that ZFC (which is generally recognized as being powerful enough to formalize all of mathematics) cannot prove that the 3 Gb string containing a MPEG2 representation of the movie "The Matrix" that is on the DVD I just bought cannot be compressed losslessly to 3 Mb as a representation as a Java program with an associated data file. Of course, this *could* be because the file actually *can* be compressed losslessly to 3 Mb, but this is hardly likely.

We can replace ZFC with Peano Arithmetic at the cost of going through more pain when representing "$K(x) \geq i$". We thus have a version of Gödel's

incompleteness theorem. This way of proving the incompleteness theorem brings up an interesting issue. We have seen that a deterministic algorithmic procedure has no way of coming up with true statements of the form "$K(x) \geq i$" for large values of i. However, there is a *randomized* procedure that can: If we pick $x \in \{0,1\}^n$ uniformly at random, the probability that "$K(x) \geq n-k$" is a false statement is less than 2^{-k}, simply because there are only $2^{n-k}-1$ binary strings of length less than $n-k$, so at most this many strings among the 2^n strings of length n can have Kolmogorov complexity less than $n-k$. Thus, given some "security parameter" k (say, k=100), consider the following way of enhancing a system such as Peano Arithmetic. We keep the axioms and inference rules of PA but add a way of generating additional axioms: We keep a counter c, initially $c = k$. At any point in time we are allowed to generate as an axiom "$K(x) \geq n - c$" (or, more precisely, the formalization of this statement within Peano arithmetic) for an x we generate uniformly at random among all $x \in \{0,1\}^n$ for any n we may choose. After having generated such an axiom, we must increment c. This system has the following interesting properties.

- The probability that we ever, throught the infinite lifetime of the system, generate an axiom that is a false statement about the integers is at most $\sum_{c=k}^{\infty} 2^{-c} = 2^{-k+1}$. If $k = 100$, this is completely negligible.
- Thus, as our rules of inference are sound we will, with probability at least $1 - 2^{-k+1}$, only generate true statements about the integers throughout the infinite lifetime of our system.
- Our system is *not* recursively enumerable; it cannot be simulated by any deterministic algorithmic procedure. However, it may be implemented if we have access to a source of randomness.

As the system is not recursively enumerable, *Gödels theorem does not apply to the system*, and we may ask if, with non-negligible probability, the system is powerful enough to generate all true statements of number theory? That this is *not* the case is a deep theorem proved by Leonid Levin. Nevertheless, the system provides an intriguing counterexample to the thesis that mathematical knowledge is always attained (or in principle attainable) by recursively enumerable computation on a recursive set of axioms.