Kolmogorov complexity ; induction, prediction and compression

Contents

Remarque : peut-être mentionner concaténation.

1 Motivation for Kolmogorov complexity

When faced with the sequence 2 4 6 8 10 12 xs, anybody would expect x to be 14. However, one could argue that the sequence 2 4 6 8 10 12 14 does not exist "more" than 2 4 6 8 10 12 13, or 2 4 6 8 10 12 0: there seems to be no reason for picking 14 instead of anything else. There is an answer to this argument: 2 4 6 8 10 12 14 is "simpler" than other sequences, in the sense that it has a shorter description.

This can be seen as a variant of Occam's razor, which states that for a given phenomenon, the simplest explanation should be preferred. This principle has been formalized in the 60s by Solomonoff and Kolmogorov: as we will soon see, the *Kolmogorov complexity* of an object is essentially the length of its shortest description, where "description of x" means "algorithm that can generate x", measured in bytes. However, in practice, finding the shortest description of an object is difficult.

Kolmogorov complexity provides a reasonable justification for "inductive reasoning", which corresponds to trying to find short descriptions for sequences of observations. The general idea is that any regularity, or structure, detected in the data can be used to compress it.

This criterion can also be used for prediction: given a sequence $x_1, ..., x_n, (?)$ choose the x_{n+1} such that the sequence $x_1, ..., x_{n+1}$ has the shortest description, or in other words, such that x_{n+1} "compresses best" with the previous x_i . For example, given the sequence 0000000?, 0 should be predicted, because 00000000 is simpler than $0000000x$ for any other x.

As a more sophisticated example, given a sequence $x_1, y_1, x_2, y_2, x_3, y_3, x_4, ...$, if we find a simple f such that $f(x_i) = y_i$, we should predict $f(x_4)$ as the next element of the sequence. With this kind of relationship, it is only necessary to know the x_i, and f to be able to write the full sequence. If we also have $x_{i+1} = g(x_i)$, then only x_0,f and g have to be known: somebody who has understood how the sequence $(1, 1; 2, 4; 3, 9; 4, 16; ...)$ is made will be able to describe it very efficiently.

Any better understanding of the data can therefore be used to find structure in the data, and consequently to compress it better: comprehension and compression are essentially the same thing.

In the sequence $(1, 1; 2, 4; 3, 9; 4, 16; ...)$, f was very simple, but the more data we have, the more complicated f can reasonably be: if you have to learn by heart two sentences x_1, y_1, where x_1 is in English, and y_1 is x_1 in German, and you do not know German, you should just learn y_1. If you have to learn by heart a very long sequence of sentences such that x_i is a sentence in English and y_i is its translation in German, then you should learn German. In other words, the more data we have, the more interesting it is to find regularity in them.

The identity between comprehension and compression is probably even clearer when we consider text encoding: with a naïve encoding, a simple text in English is around 5 bits for each character (26 letters, space, dot), whereas the best compression algorithms manage around 3 bits per character. However, by removing random letters from a text and having people try to read it, the actual information has been estimated at around 1.1 bits per character.

2 Formal Definition

Let us now define the Kolmogorov complexity formally:

Definition 1. *The Kolmogorov complexity of x a sequence of 0s and 1s is by definition the length of the shortest program on a Universal Turing machine*[1] *that prints x. It is measured in bits.*[2]

The two propositions below must be seen as a sanity check for our definition of Kolmogorov complexity.

Proposition 2 (Kolmogorov complexity is well-defined)**.** *The Kolmogorov complexity of x does not depend on the Turing machine, up to a constant which does* not *depend on x (but does depend on the two Turing machines).*

Sketch of the proof. if P_1 prints x for the Turing machine T_1, then if I_{12} is an interpreter for language 1 in language 2, $I_{12} :: P_1$ prints x for the Turing machine T_2, and therefore $K_2(x) \leqslant K_1(x) + \text{length}(I_{12})$ □

[1] Your favorite programming language, for example.

[2] Or any multiple of bits.

In other words, if P is the shortest zipped program that prints x in your favourite programming language, you can think about the Kolmogorov complexity of x as the size (as a file on your computer) of P (we compress the program to reduce differences from alphabet and syntax between programming languages).

If the objects we are interested in are not sequences of 0 and 1 (pictures, for example), they have to be encoded.

Proposition 3. *the Kolmogorov complexity of x does not depend on the encoding of x, up to a constant which does* not *depend on x (but does depend on the two encodings).*

Sketch of the proof. Let f,g be two encodings of x. We have $K(g(x)) < K(f(x))+ K(g \circ f^{-1})$ (instead of encoding $g(x)$, encode $f(x)$, and the map $g \circ f^{-1}$. $\max\left(K(g \circ f^{-1}), K(f \circ g^{-1})\right)$ is the constant). □

In other words, the Kolmogorov complexity of a picture will not change if you decide to put the most significant bytes for your pixels on the right instead of the left.

Notice that the constants for these two theorems are reasonably small (the order of magnitude should not exceed the megabyte, while it is possible to work with gigabytes of data).

3 Trying to compute Kolmogorov complexity

Kolmogorov complexity is not computable. Even worse, it is never possible to prove that the Kolmogorov complexity of an object is large.

An intuitive reason for the former fact is that to find the Kolmogorov complexity of x, we should run all possible programs in parallel, and choose the shortest program that outputs x, but we do not know when we have to stop: There may be a short program still running that will eventually output x. In other words, it is possible to have a program of length $K(x)$ that outputs x, but it is *not* possible to be sure that it is the shortest one.

Theorem 4 (Chaitin's theorem)**.** *There exists a constant L[3] such that it is not possible to prove the statement $K(x) > L$ for any x.*

Sketch of the proof. For some L, write a program that tries to prove a statement of the form $K(x) > L$ (by enumerating all possible proofs). When a proof of $K(x) > L$ for some x is found, print x and stop.

If there exists x such that a proof of $K(x) > L$ exists, then the program will stop and print some x_0, but if L has been chosen largen enough, the length of the program is less than L, and describes x_0. Therefore, $K(x_0) \leqslant L$. contradiction. □

The link with Berry's paradox is clear:
"The smallest number that cannot be described in less than 13 words"
"The [first x found] that cannot be described in less than [L] bits".

[3]reasonably small, around 1Mb

Corollary 5. *Kolmogorov complexity is not computable.*

Proof. Between 1 and 2^{L+1}, there must be at least one integer n_0 with Kolmogorov complexity greater that L (since there are only $2^{L+1}-1$ programs of length L or less). If there was a program that could output the Kolmogorov complexity of its input, we could prove that $K(n_0) > L$, which contradicts Chaitin's theorem.

□

As a possible solution to this problem, we could define $K_t(x)$, the length of the smallest program that outputs x in less than t instructions, but we lose some theoretical properties of Kolmogorov complexity (Proposition 2 and 3 have to be adapted to limited time but they are weaker, see [4], Section 7. For example, Proposition 2 becomes $K_{ct\log_2 t,1}(x) \leqslant K_{t,2}(x)+c)$, for no practical gain.

4 Standard upper bounds

Because of Chaitin's theorem, the best we can do is finding upper bounds on Kolmogorov complexity. First, we introduce *prefix-free* Kolmogorov complexity and prove *Kraft's inequality*, and we then give classical upper bounds, firstly for integers, then for other strings.

Remember that because of Proposition 2, all inequalities concerning Kolmogorov complexity are true *up to an additive constant*. We consequently write $K(x)\overset{+}{\leqslant} f(x)$ for $K(x) \leqslant f(x) + a$, where a does not depend on x.

4.1 Encoding integers, prefix-free complexity

If we simply decide to encode an integer by its binary decomposition, then, we do not know for example if the string "10" is the code for 2, or the code for 1 followed by the code for 0.

Similarly, given a Turing machine, and two programs P_1 and P_2, there is nothing that prevents the concatenation P_1P_2 from defining another program that might have nothing to do with P_1 or P_2.

This leads us to the following (not formal) definition:

Definition 6. *A set of strings S is said to be* prefix-free *is no element of S is a prefix of another.*

A code is said to be prefix-free *if no codeword is a prefix of another (or equivalently, if the set of codewords is prefix-free).*

We then adapt our definition of Kolmogorov complexity by forcing the set of programs to be prefix-free (hence the name *prefix-free* Kolmogorov complexity[4]).

It is clear now that if we receive a message encoded with a prefix-free code, we can decode it unambiguously letter by letter, while we are reading it, and if a Turing machine is given two programs P_1 and P_2, their concatenation will not be ambiguous.

It is important to notice that in practice, when thinking with programming languages, working with prefix-free complexity does not change anything, since

[4] "self-delimited" is sometimes used instead of prefix-free.

the set of compiling programs is prefix free (the compiler is able to stop when the program is finished).

Now, let us go back to integers:

Proposition 7. *Let n be an integer. We have*

$$K(n) \overset{+}{\leqslant} \log_2 n + 2\log_2 \log_2 n. \tag{1}$$

Proof, and a bit further. Let n be an integer, and let us denote by b its binary expansion, and by l the length of its binary expansion (i.e. $l = \lfloor \log_2(n+1) \rfloor \sim \log_2 n$).

Consider the following prefix codes (if c is a code, we will denote by $c(n)$ the codeword used for n):

- c_1: Encode n as a sequence of n ones, followed by zero. Complexity n.
- c_2: Encode n as $c_1(l) :: b$, with l and b as above. To decode, simply count the number k of ones before the first zero. The k bits following it are the binary expansion of n. Complexity $2\log_2 n$.
- c_3: Encode n as $c_2(l) :: b$. To decode, count the number k_1 of ones before the first zero, the k_2 bits following it are the binary expansion of l, and the l bits after these are the binary expansion of n. Complexity $\log_2 n + 2\log_2 \log_2 n$, which is what we wanted.
- We can define a prefix-free code c_i by setting $c_i(n) = c_{i-1}(l) :: b$. The complexity improves, but we get a constant term for small integers, and anyway, all the corresponding bounds are still equivalent to $\log_2 n$ as $n \to \infty$.
- It is easy to see that the codes above satisfy $c_n(1) = n + 1$: for small integers, it would be better to stop the encoding once a number of length one (i.e. one or zero) is written. Formally, this can be done the following way: consider $c_{\infty,1}$ defined recursively as follows :

$$c_{\infty,1}(n) = c_{\infty,1}(l-1) :: b :: 0, \tag{2}$$
$$c_{\infty,1}(1) = 0. \tag{3}$$

  It is easy to see that $c_{\infty,1}(n)$ begins with 0 for all n, i.e, $c_{\infty,1} = 0 :: c_{\infty,2}$.

  We can now set $c_\infty(0) = 0$ and $c_\infty(n) = c_{\infty,2}(n+1)$.

The codes c_2 and c_3 are similar to Elias gamma and delta (respectively) coding. c_∞ is called Elias omega coding.

□

4.2 Classical bounds

We also have the following bounds:

1. A simple program that prints x is simply print(x). The length of this program is the length of the function print, plus the length of x, but it is also necessary to provide the length of x to know when the print

function stops reading its input (because we are working with prefix-free Kolmogorov complexity). Consequently

$$K(x) \overset{+}{\leqslant} |x| + K(|x|). \tag{4}$$

By counting arguments, some strings x have a Kolmogorov complexity larger than $|x|$. These strings are called *random* strings by Kolmogorov. The justification for this terminology is that a string that cannot be compressed is a string with no regularities.

2. The Kolmogorov complexity of x is the length of x compressed with the *best* compressor for x. Consequently, we can try to approximate it with any standard compressor, like zip, and we have:

$$K(x) \overset{+}{\leqslant} |\text{zip}(x)| + |\text{unzip\_program}|. \tag{5}$$

This property has been used to define the following distance between two objects: $d(x,y) = \frac{\max(K(x)|y), K(y|x))}{\max(K(x), K(y))}$. By using distance-based clustering algorithms, the authors of [1] have been able to cluster data (MIDI files, texts...) almost as anyone would expected (the MIDI files were clustered together, with subclusters essentially corresponding to the composer, for example). In the same article, the Universal Declaration of Human Rights in different languages has been used to build a satisfying language tree.

3. If we have some finite set E such that $x \in E$, then we can simply enumerate all the elements of E. In that case, an x can be described as "the n^{th} element of E. For this, we need $K(E)$ bits to describe E, and $\lceil \log_2 |E| \rceil$ bits to identify x in E:

$$K(x) \overset{+}{\leqslant} K(E) + \lceil \log_2 |E| \rceil. \tag{6}$$

4. More generally, if μ is a probability distribution on a set X, and $x \in X$, we have

$$K(x) \overset{+}{\leqslant} K(\mu) - \log_2(\mu(x)). \tag{7}$$

For example, if μ is uniform, we find equation (6). Another simple case is the i.i.d. case, which will be discussed later. This inequality is the most important one for machine learning (it is in fact the reason for the $\log_2(\mu(x))$ often found in machine learning), and will be proved below. Notice that (7) is true for *any* probability distribution on X, but the bound is tighter if both μ is "simple", and $\mu(x)$ is high. Our goal is therefore to find such distributions.

The idea behind (7) is that for a given set E, if I think some elements are more likely to appear than others, they should be encoded with fewer bits. For example, if in the set $\{A, B, C, D\}$, we have $P(A) = 0.5$, $P(B) = 0.25$, and $P(C) = P(D) = 0.125$, instead of using a uniform code (two bits for each character), it is better to encode for example[5] A with 1, B with 01, C with 001 and D with 000.

[5] We do not care about the code, we care about the length of the code words for the different elements of X.

In the first example, the expected length with the uniform code is 2 bits per character, while it is 1.75 bits per character for the other.

In general, it can be checked that the expected length is minimal if the length of the code word for x is $-\log_2(\mu(x))$. If we have a code satisfying this property, then (7) follows immediately (encode μ, and then use the code corresponding to μ to encode x).

However, if we stick to encoding one symbol after another approximations have to be made, because we can only have integer codelengths. For example, consider we have: $P(A) = 0.4$, $P(B) = P(C) = P(D) = P(E) = 0.15$. The $-\log_2 P(*)$ are not integers: we have to assign close integer codelengths. We describe two possible ways of doing this:

- Sort all symbols by descending frequency, cut when the cumulative frequency is closest to 0.5. The codes for the symbols on the left (resp. right) start with 0 (resp. 1). Repeat until all symbols have been separated. This is Shannon–Fano coding.
- Build a binary tree the following way: Start with leave nodes corresponding to the symbols, with a weight equal to their probability. Then, take the two nodes without parents with lowest weight, and create a parent node for them, and assign it the sum of its children's wieght. Repeat until only one node remains. Then, code each symbol with the sequence of moves from the root to the leaf (0 corresponds to taking the left child, for example). This is Huffman coding, which is better than Shannon–Fano coding.

On the example above, we can find the following codes (notice that some conventions are needed to obtain well-defined algorithms from what is described above: for Shannon-Fano, what to do when there are two possible cuts, and for Huffman, which node is the left child and which node is the right child):

	Theoretical optimal length	Shannon–Fano code	Huffman code
A	≈ 1.322	00	0
B	≈ 2.737	01	100
C	≈ 2.737	10	101
D	≈ 2.737	110	110
E	≈ 2.737	111	111
Length expectation	≈ 2.17	2.3	2.2

As we can see, neither Shannon–Fano coding nor Huffman coding reach the optimal bound.

However, if instead of encoding each symbol separately, we encode the whole message (7) can actually be achieved up to a constant number of bits for *the whole message*[6] by describing a simplification of arithmetic coding:

The idea behind arithmetic coding is to encode the whole message as a number in the interval $[0, 1]$, determined as follow: consider we have the message $(x_1, ..., x_N) \in X^N$ (here, to make things easier, we fix N). We start with the full interval, and we partition it in $\#X$ subintervals, each one corresponding to

[6]Since all the inequalities were already up to an additive constant, this does not matter at all.

some $x \in X$ and of length our expected probability to see x, given the characters we have already seen, and we repeat until the whole message is read. We are left with an subinterval I_N of $[0,1]$, and we can send the binary expansion of any number in I_N (so we choose the shortest one).

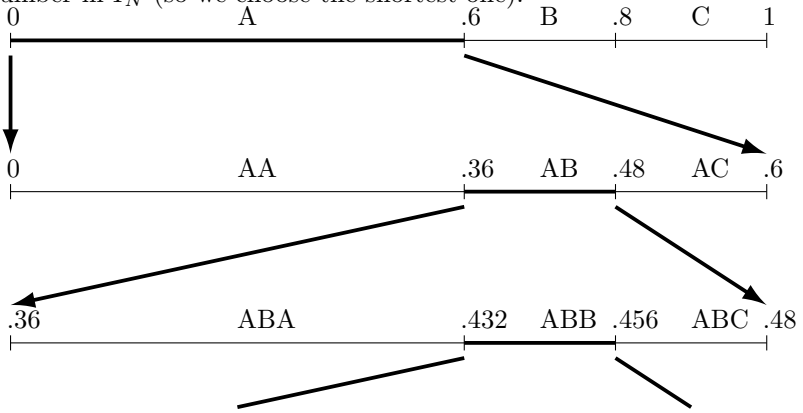

Figure 1: Arithmetic coding of a word starting with ABB, with $P(A) = 0.6$, $P(B) = 0.2$, $P(C) = 0.2$

Algorithm 8 (End of the proof of (7): A simplification of arithmetic coding)**.** *We are given an ordered set X, and for x_1,...,x_n,$y \in X$, we denote by $\mu(y|x_1, ..., x_n)$ our expected probability to see y after having observed $x_1, ..., x_n$.*[7]

Goal: *Encode the whole message in one number $0 \leqslant x \leqslant 1$.*

Encoding algorithm

Part 1*: Encode the message as an interval.*

$i = 1$
$I = [0, 1]$
while $x_{i+1} \neq$ END ***do***
Partition I into subintervals $(I_x)_{x \in X}$ such that:
$x < y \implies I_x < I_y$,[8] $\text{length}(I_x) = \text{length}(I)\mu(x|x_1, ..., x_{i-1})$
Observe x_i
$I = I_{x_i}$
$i = i + 1$
end while
return I

Part 2*: pick a number in the interval I.*

We can find a binary representation for any real number $x \in [0, 1]$, by writing $x = \sum_1^{+\infty} \frac{a_i}{2^i}$, with $a_i \in \{0, 1\}$. Now, for a given interval I, pick the number $x_I \in I$

[7] A simple particular case is the case where μ does not depend on past observations (i.e. $\mu(y|x_1, ..., x_n) =: \mu(y)$) and can therefore be identified with a probability distribution on X

[8] If I, J are intervals, we write $I < J$ for $\forall x \in I$, $\forall y \in J$, $x < y$.

which has the shortest binary representation.[9] *The message will be encoded as* x_I.

Decoding algorithm. x_I *received.*
$i = 1$
$I = [0, 1]$
while *not termination criterion*[10] ***do***
Partition I *into subintervals* $(I_x)_{x \in X}$ *as in the encoding algorithm.*
$x_i \leftarrow$ *the only* y *such that* $x_I \in I_y$
$I = I_{x_i}$
$i = i + 1$
end while

Arithmetic coding allows to find a code such that the length of the code word for $x = (x_1, ..., x_n)$ is $\sum_{i=1}^{n} - \log_2(\mu(x_i|x_1, ..., x_{i-1})) = \sum_{i=1}^{n} - \log_2(\mu(x_i|x_1, ..., x_{i-1})$, which is what we wanted.

However, arithmetic cannot be implemented like this, because of problems of finite precision: if the message is too long, the intervals of the partition might become undistinguishable. It is possible to give an online version of this algorithm which solves this problem: the encoder sends a bit once he knows it (he can send the n-th bit if I contains no multiple of 2^{-n}). In that case, he does not have to worry about the n first digits of the bounds of the intervals of the partition anymore, which solves the problem of finite precision. In the case 0.5 remains in the interval, the encoder can remember that if the first bit is 0, then the second is 1, and conversely, to work in $[.25, .75]$ instead of $[0, 1]$ (and the process can be repeated).

As an interesting particular case of equation (7), let us consider the i.i.d. case on X^n for equation (7): for $x_1, ..., x_n \in X$, $\mu(x_1, ..., x_n) = \alpha(x_1)...\alpha(x_n)$. In that case, the right side of equation (7) becomes:

$$K(n) + K(\alpha) - \sum_{i=1}^{n} \log_2 \alpha(x_i). \tag{8}$$

[9] We call *length of the representation* $(a_1, ..., a_n, ...)$ the number $\min\{n \in \mathbb{N}, \forall k > n, a_k = 0\}$. If $\text{length}(I) \neq 0$, I necessarily contains number with a finite representation.

[10] There are several possibilities for the termination criterion:

- We can have an END symbol, to mark the end of the message (in that case, stop when END has been decoded).
- We can give the length N of the message first (in that case, stop when N characters have been decoded)
- The message sent by the encoder can be slightly modified as follows. We decide that a sequence of bits $(a_1, ..., a_n)$ represents the interval I_a of all numbers of which the binary expansion starts with $(a_1, ..., a_n)$. Now, the message a sent by the encoder is the shortest non ambiguous message, in the sense that I_a is contained in I, but is not contained in any of the subintervals I_x corresponding to I (where I is the interval obtained at the end of the first part of the encoding algorithm).

of which the expectation is equal to (if the x_i are really sampled from α):

$$K(n) + K(\alpha) - n \sum_{x \in X} \alpha(x) \log_2 \alpha(x). \tag{9}$$

In particular, once α is described, we need $-\sum_{x \in X} \alpha(x) \log_2 \alpha(x)$ bits (in expectation) to encode x_i.

If the x_i are actually sampled from a probability distribution β, but we use a code adapted for the distribution α, we will need more bits to encode our message. The expecation of the number of additional bits needed to encode $(x_1, ..., x_n)$ is:

$$n \sum_{x \in X} \beta(x)(\log_2 \beta(x) - \log_2 \alpha(x)) = n \sum_{x \in X} \beta(x) \log_2 \frac{\beta(x)}{\alpha(x)}. \tag{10}$$

Definition 9. *The quantity* $H(\mu) = -\sum_{x \in X} \mu(x) \log_2 \mu(x)$ *is called* entropy *of* μ.

The quantity $\mathrm{KL}(\mu \| \nu) := \sum_{x \in X} \nu(x) \log_2 \frac{\nu(x)}{\mu(x)}$ *is called* Kullback-Leibler divergence *from* ν *to* μ.

4.3 Link with model selection

The term $-\log_2(\mu(x))$ is essentially the cost of encoding the data with the help of the model, whereas $K(\mu)$ can be seen as a penalty for complicated models (which, in machine learning, prevents overfitting: if the data is "more compressed", including the cost of the model and the decoder, the description is better).

As a basic example, if μ is the Dirac distribution at x, $K(\mu) = K(x)$: in that case, all the complexiy is in the model, and the encoding of the data is completely free.

More interestingly, if we are trying to fit the data $x_1, ..., x_n$ to an i.i.d. Gaussian model (which corresponds to the description: "this is Gaussian noise"), with mean m and fixed variance σ^2, the term $-\log_2(\mu(x))$ is equal to $\sum_i (x_i - m)^2$ up to additive constants, and the m we should select is the solution to this least square problem, which happens to be the sample mean (if we neglect $K(m)$, which corresponds to finding the maximum likelihood estimate).

In general, $K(\mu)$ is difficult to evaluate. There exists two classical approximation, yielding different results:

- $K(\mu)$ can be approximated by the number of parameters in μ. This gives the Akaike Information Criterion (AIC).
- $K(\mu)$ can also be approximated by half the number of parameters in μ multiplied by the logarithm of the number of observations. This gives the Bayesian Information Criterion (BIC). The reason for this approximation will be given in Talk 2.

5 Possible approximations

Now, even with these upper bounds, in practice, it is difficult to find good programs for the data.

As an introduction to the next talk, we give some heuristics:

- Usual compression techniques (like zip) can yield good results.
- Minimum description length techniques: starting from naive generative models to obtain more complex ones: for example, if we have to predict sequences of 0 and 1, and we initially have two experts, one always predicting 0 and the other predicting always 1, we can use a mixture of experts: use the first experts with probability p, and the second with probability $1-p$, and we can obtain all Bernoulli distributions. More interestingly, we can also automatically obtain strategies of the form "after having observed x_i, use expert k to predict x_{i+1}", thus obtaining Markov models.
- Auto-encoders can also be used: they are hourglass shaped neural networks (fewer nodes in the intermediate layer), trained to output exactly the input. In that case, the intermediate layer is a compressed form of the data, and the encoder and decoder are given by the network.
- The model class of Turing machine is very large. For example, if we restrict ourselves to finite automata, we can compute the restricted Kolmogorov complexity, and if we restrict ourselves to visible Markov models we can even use Kolmogorov complexity for prediction.

References

[1] Rudi Cilibrasi and Paul Vitanyi. Clustering by compression.

[2] Peter D. Grünwald. *The Minimum Description Length Principle (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[3] Marcus Hutter. On universal prediction and Bayesian confirmation. *Theoretical Computer Science*, 384(1):33–48, 2007.

[4] Ming Li and Paul M.B. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition, 2008.

[5] Ray J. Solomonoff. A formal theory of inductive inference. *Information and Control*, 7, 1964.