Abrahim Ladha

Kolmogorov Complexity — Lecture 16

Consider the following two strings:

111111 1111 11011111101

definability vs description

We would describe the first string as simple. It has a relatively short description of just its length. A description of the second string must contain information about the location of the two zeroes. If you were to describe it to someone, your description should contain this information. We would like to call it complex, but we will formalize this notion.

Aside

A measure is a function μ : things $\Rightarrow \mathbb{R}^+$ (or $\mathbb{N}$) where $\mu(x) = 0 \Rightarrow x$ has nothing "none of the thing". $\mu(x) > \mu(y) \Rightarrow$ "x has more of it than y". Examples of measures include length, area, and volume. Cardinality is a measure of the sizes of sets.

We want to construct a measure on strings for their "algorithmic complexity" Given a string, how easy or hard is it to describe? Is it simple or complex? How much information does the string communicate? Can we even measure this? It turns out, yes we can. Define $K : \Sigma^* \to \mathbb{N}$ to be the Kolmogorov complexity of a string. $K(x)$ = "the length of the shortest program to print x and halt."

$$K(x) = \min_{p \in \Pi} \{ |p| : U(p, \varepsilon) = x \}$$

- x: string
- p: program
- Π: set of all programs
- $|p|$: length of program like a string
- U: universal simulator runs p on input ε
- p: program
- ε: takes no input
- x: prints x and halts; outputs x

$$K(x) = \min_{p \in \Pi} \{ |p| : U(p, \varepsilon) = x \}$$

Why did I say program and not Turing machine? Like asymptotic analysis in the theory of algorithms, our complexity measure is independent of this selection of language. Lets ~~proof~~ prove it.

Consider two language specific definitions K_{py}, K_{rust} defined like you might expect. By the Church-Turing Thesis, since rust is Turing-complete, you certainly could write a python interpreter in rust. Let this program written in rust to interpret python be called $\Pi_{py\ in\ rust}$. Now given any python program, combined with our interpreter in rust, that's just a rust program! Suppose $\exists p$ with $|p| = K_{py}(x)$ This python program, we can use as a rust program. p prints x, so we construct a rust program to print x by interpreting p. we see then that

$$K_{rust}(x) \leq K_{py}(x) + |\Pi_{py\ in\ rust}|$$

we can only say $\leq$ instead of $=$ since we do not know if there exists a smaller rust program. Notice by the Church-Turing Thesis that python is also Turing-complete. So you can write a rust interpreter in python! By a symmetrical argument, $\exists$ a rust interpreter written in python named $\Pi_{rust\ in\ py}$ and we observe

$$K_{py}(x) \leq K_{rust}(x) + |\Pi_{Rust\ in\ py}|$$

Notice that our interpreters are independent of anything about x, its complexity or its length. They are of constant size. Next notice our two inequalities are symmetric. For any two $a, b \in \mathbb{N}$, if $a \leq b + O(1)$ and $b \leq a + O(1)$, then $|a - b| \leq O(1)$. we combine our inequalities this way to get that $\exists$ c such that

$$\forall x \quad |K_{py}(x) - K_{rust}(x)| \leq c$$

So the difference between our two algorithmic complexities only differs by some constant! (TC) This can obviously be generalized for all programming languages so we drop the subscript and consider $K(x)$.

Now we prove some fun facts, to rigorously understand the our measure of descriptive complexity better. Notice for any $x \in \Sigma^*$, there exists a program to print it

```
def f():
    x = "                                                    "
    print(x)
```

Consider the program with some hardcoded string. We can see that $\forall x\ K(x) \leq |x| + c$ where c is some constant independent of the input. for example $|$"print()"$| = 7$ so $c \geq 7$.

What about $K(xx)$? What is the Kolmogorov complexity of some string concatenated with itself twice. A bael idea is to hardcode xx. Notice that xx has some internal, less random looking structure. Rather than an upper bound of $K(xx) \leq |xx| + c = 2|x| + c$, we can construct a program to only store x (not xx) and then compute xx from x. This gives us a better upper bound of $K(xx) \leq |x| + c'$. Here $c' > c$ since our program needs the logic to compute xx (print(x.append(x))) Its still a constant though. Its worth mentioning once, but I wont make this remark in the future. What about many concatenations? What is $K(x^n)$ for some n? We can immediately generalize our idea. Our program as a function of the input is $|x|$ and $|n| = \log n$. So $K(x^n) \leq |x| + \log n + c$. We also need not hardcode x, it may not be so random looking, so even $K(x^n) \leq K(x) + \log n + c$ where we replace hardcoded x by a program to compute x.

```
def f():
    x = "                    "
    n = "                    "
    for i in range(n)
        answer = answer.append(x)
```

For x^R the reversal of string x, what is $K(x^R)$ relative to $K(x)$? Well if some program p prints x, we can make a program q to print x^R. q is like p, it computes x but instead of printing it, it reverses it, then prints it. We observe this reversal operation is independent of the input, so $|q| = |p| + c$. We see that $|K(x) - K(x^R)| \leq c$ for some constant c.

x, x^R invariant to "simple".

Lets go back to some intuition about randomness. Some strings appear to have very short simple descriptions, like 1^n. Some appear to have long descriptions, or at least no short ones. We say a string is incompressible if $K(x) \geq |x| - c$. There isn't a much shorter description than just saying the string. We say a string is compressible if it is not incompressible.

How many strings of length n are compressible by two bits? Just two bits. Lets compute it as a ratio:

$$\frac{|\{x \in \Sigma^n \mid K(x) \leq |x| - 2\}|}{|\Sigma^n|} \leq \frac{|\{p \in \Pi \mid p \text{ a program with } |p| \leq n-2\}|}{2^n} \leq$$

$$\frac{\bigcup_{i=0}^{n-2} \Sigma^i}{2^n} \leq \frac{\sum_{i=0}^{n-2} 2^i}{2^n} = \frac{2^{n-1}}{2^n} = \frac{1}{2}.$$

Half. HALF? Only half of the strings of length n are compressible by two bits. This generalizes so only a quarter are compressible by three bits, and $1/2^{d-1}$ are compressible by d bits. This is a very lazy upper bound, but its still sufficient to show us most strings are incompressible. 1/1000 strings are compressible by 11 bits. I want to stress the "most" part. We have found a deep connection between randomness and information content. A uniformly random string has overwhelming probability to be incompressible. The compressible strings are the lucky ones.

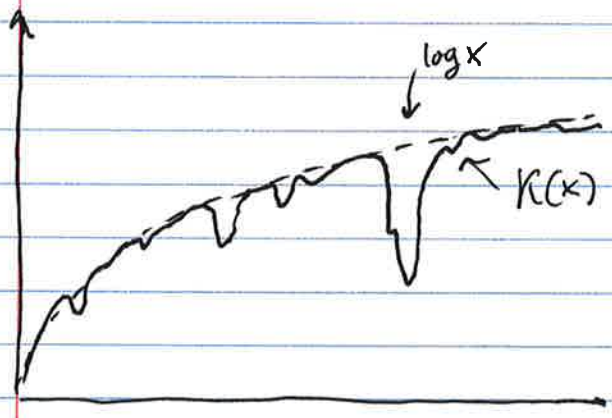
Let us try to understand more about what $K(x)$ looks like. Consider the plot of $K(x)$, but instead of $K: \Sigma^* \to \mathbb{N}$, consider it as a natural function $K: \mathbb{N} \to \mathbb{N}$. First note that $K(x)$ grows unbounded. $\nexists c$ s.t. $\forall x \in \Sigma^* \; K(x) < c$. The proof is obvious, intuitively longer strings are probably more complex. Assume to the contrary some such c exists. Choose n such that $K(1^n) > c$. n can get really big, c cannot. So we see the function is growing. A second fact, observe that $K(x)$ "hugs" $\log x$. Why? Because we proved most strings are incompressible, it not not often dip below $\log x$. A third fact is that even if it does hug $\log x$, it should dip below infinitely often. Recall that we showed $K(x) \approx K(x^R)$. ~~Our int~~ This matches our intuition. A description of a string being simple or complex should be invariant to which way we read it, forwards or backwards. By a similar argument, we see that $K(x) \approx K(2x) \approx K(3x) \approx K(2^x) \approx K(2^{2^x}) \approx K(x+\sqrt{x})$ and so on. A ~~minimal~~ small program for one string implies ~~a small program~~ infinite family of small programs, for an infinite family of strings. Our final remark is that $K(x)$ can be analogous to a continuity of a real valued function. Recall that a function is continuous if when $x, x+\varepsilon$ are chose, so is $f(x), f(x+\varepsilon)$. That is, $|x - x_0| < c_1 \Rightarrow |f(x) - f(x_0)| < c_2$. Similarly, there exists some c such that $\forall x \; |K(x) - K(x \pm 1)| < c$. Take the program which prints x. Modify it to add one. Now you have a program to print $x+1$. I claim such a plot of $K(x)$ may look like:

grows

hugs $\log x$

dips infinitely and often

continuous



All this discussion on what $K(x)$ looks and feels like, we never even gave an algorithm for it. That's because none exists. $K(x)$ is not a computable function. We proceed by a kind of diagonalization. Assume to the contrary that $K(x)$ is computable. Then we may construct the following algorithm. On input w, it searches for the smallest lexographic string with Kolmogorov complexity greater than the length of w. Here

```
M on input w:
    for x ∈ Σ* lexographically:
        if K(x) > |w|
            print x
            halt
```

The for loop iterates like $x \in \{\varepsilon, 0, 1, 00, 01, \ldots\}$. What is M on input $\langle M \rangle$? What is $M(\langle M \rangle)$? Well as the algorithm proceeds, $M(\langle M \rangle)$ will search for some x such that $K(x) > |\langle M \rangle|$ and print it. But since M itself prints this x, we see $K(x) \leq |\langle M \rangle|$. A contradiction. The statements $a > b$ and $a \leq b$ cannot both be true. We see then that $K(x)$ is not computable. This proof is quite simple, but comes at the cost of having a bit of a rough edge.

mention recursion theorem

Kolmogorov complexity also gives us an extremely useful proof technique known as the method of incompressibility. It shows up too many places for us to give a complete picture of its applications. Instead we only give two.

We prove that there exists infinitely many primes. Suppose not that there exists only finitely many, $p_1, \ldots, p_k$. Then $\forall n \in \mathbb{N}$ $\exists e_1 \ldots e_k$ such that $n = p_1^{e_1} \cdots p_k^{e_k}$. Then $\langle e_1, \ldots, e_k \rangle$ is a description of n. Note then each e_i can be described in $\log e_i$ bits, so $K(n) \leq \log(e_1) + \ldots \log(e_k)$. Also notice each $e_i < \log n$, so $K(n) \leq \log(e_1) + \cdots + \log(e_k) \leq k \log\log n$ where k is independent of the input n. So $K(n) \leq O(\log\log n)$. (Choose any incompressible n and we have a contradiction.

We were able to show that if there were finitely many primes, there exists a succinct description of all numbers. But we know most are incompressible. We may choose any incompressible n to reach a contradiction. We concluded $\forall n$ $K(n)$ is $O(\log \log n)$, but we know for most n that $K(n)$ is $\Omega(\log n)$.

You may think of the incompressibility method in two ways. First assume something to the contrary. This assumption leads to succinct (ie, compressible) descriptions of some kinds of objects. Existence of incompressible objects then leads to a contradiction. Another way is analogous to the pigeonhole principle or the probabilistic method. Those methods show existence of an object with desirable properties. The method of incompressibility shows most objects have a desired property. "Most" here means truly in a Kolmogorov-random sense. It is one of the strongest techniques we have for average case and worst case lower bounds.

Lets use the method of incompressibility to prove some languages are not regular. We prove what I call the extremely weak KC regularity lemma. The book proves a generalization called the KC regularity lemma. It also proves a second generalization called the KC-regularity characterization. I have limited the power here because I want as simple a tool as possible to prove languages are not regular.

(extremely weak) KC-regularity lemma: Assume L is regular. Let $xy \in L$ where $|xy|$ is some function of some n, and [redacted]. Then $K(y)$ is $O(1)$. Lets prove it.

↑ y is the smallest string such that $xy \in L$. ($xy' \in L$ with longer y' may exist)

proof

If L is regular, $\exists$ a DFA D for it. Run x on D to get to some state q_i. Since y is the minimal suffix of xy ($\exists y'$ $xy' \in L$) Then y is the minimal string to take DFA D from state q_i to an accept state. Then D, q_i, and this discussion are a unique description of y. So it follows $K(y) \leq |D| + |q_i| + c$. Since the F in DFA is for finite, $K(y) \leq |D| + |q_i| + c \leq O(1)$.

recipe

We use our weak lemma to prove languages are not regular.

1. Assume to the contrary L is regular
2. choose some $xy \in L$ with $|xy|$ a function of n, it just can't be constant, rather an infinite family of strings
3. Choose x so that y is minimal and $|y|$ is also not a constant.
4. Apply the lemma to get $K(y)$ is $O(1)$.
5. But as y is defined, it should be $\omega(1)$.
6. Conclude by contradiction.

$a^n b^n$

Heres an example. Assume to the contrary $L = \{a^n b^n \mid n \in \mathbb{N}\}$ is regular. Choose $xy = a^n b^n$ and $x = a^n$. Notice then $y = b^n$ and is minimal. By the lemma, $K(b^n) = O(1)$, but since n grows unbounded, we have a contradiction.

see how fast that was! lets do some more.

1^{n^2}

Assume to the contrary $L = \{1^{n^2} \mid n \in \mathbb{N}\}$ is regular. Choose $xy = 1^{(n+1)^2}$ and $x = 1^{n^2}$. Then $y = 1^{(n+1)^2 - n^2} = 1^{2n+1}$. Notice y is minimal. By our lemma, $K(1^{2n+1}) = O(1)$, a similar contradiction.

ww^R

Assume to the contrary $L = \{ww^R \mid w \in S^*\}$ is regular. Choose $xy = (ab)^n(ba)^n$ and $x = (ab)^n$. Notice $y = (ba)^n$ is minimal, so $K((ba)^n) = O(1)$ by our lemma, a contradiction.

This proof looks easy but it hides a ton of details. If $xy = a^n a^n$, what would be the minimal y which also ~~is~~ isn't ~~a~~ a constant?, for $x = a^n$. Very very difficult to say. Choose a good xy so you never have to think about it. The not-so-weak version of the lemma is more difficult, but makes this clearer.

1^p

A final proof: Assume to the contrary $L = \{1^p \mid p \text{ is prime}\}$ is regular. Choose $xy = 1^p$ where p is the $(k+1)^{th}$ prime. Choose x such that $x = 1^{p'}$ where p' is the k^{th} prime. Notice $y = 1^{p-p'}$ is minimal. By our lemma, $K(1^{p-p'}) = O(1)$ but the difference between primes grows unbounded. A contradiction.

I will conclude with a final analogy between Kolmogorov complexity and computational learning theory. Suppose we loosened our definition of $K(x)$ so that ~~the~~ the programs to print x are allowed to make some errors. Recall our Definition of $K(x)$ = "the length of the shortest program which prints string x". Suppose I made the following synonym substitutions.

length	→	size
shortest	→	simplest
program	→	description
prints	→	approximates
string	→	dataset.

Our definition is now $K(x)$ = "the size of the simplest description which approximates dataset x". Hey! That sounds like Occam's Razor! You would be correct. Following this logic, you could formalize Occam's Razor under PAC learning. Also in practice since $K(x)$ is not computable, there is much more success with computable restrictions, like $K^t(x) = \min_{p \in \Pi} \{|p| : U(p, \varepsilon) = x \text{ halts in } t(|x|) \text{ steps}\}$