CS 4510 Automata and Complexity 03/13/2024

Lecture 17: Kolmogorov Complexity

Lecturer: Abrahim Ladha *Scribe(s): Michael Wechsler*

1 Introduction

Consider the following three strings:

111111111111111111 1111101111111111101 1011011111100001010

Lets understand our own intuition first about what objects appear simple or complex. The first string can be described simply. It has a relatively short description of just its length. The second string is less simple. Maybe we couldn't call it complex, necessarily, but it is certainly less simple. If you were to describe it, your description would also have to include information about the location of the two zeroes. The third string, at a first glance appears complex. But it is actually the trailing decimal of $e = 2.718...$ Although it may appear complex, this is actually a simple description of it.

Definition 1.1 (measure)**.** Intuitively, a measure is a function $\mu : \{\text{things}\} \to \mathbb{R}^+$ (or $\mathbb{N}$), where $\mu(x) = 0 \implies x$ has nothing, or none of "it". If $\mu(x) > \mu(y)$, then x has more of "it" than y.

Examples of measures include length, area, volume, for their respective objects. Cardinality is a measure on sets. Every probability distribution is a measure, where the measure of the whole space is 1. Entropy and temperature can also be considered measures.

2 Definition

We want to construct a measure on strings for their "algorithmic complexity." Given a string, how hard is it to describe? Is it simple or complex? How much information does the string communicate? Can we even measure this? We want to create a formal definition which captures the intuition of what it means for a finite sized object to appear random or not. What is a "description" anyway? Lets follow our intuition towards a formalization. A program is a description! This leads us to an intuitive definition.

Definition 2.1. Let $K : \Sigma^* \to \mathbb{N}$ to be the Kolmogorov Complexity of a string.

$$K(x) = \text{the length of the shortest program to print } x \text{ and halt.}$$

Mathematically, this could be represented as

$$K(x) = \min_{p \in \Pi} \Big(|p| : U(p, \varepsilon) = x \Big) \tag{1}$$

x : the string	U : universal simulator (runs p on ε)
p : a program	p : program
Π : set of all programs	ε : takes no input
$\|p\|$: length of program (as a string)	x : prints x and halts; output x

We will explore why our intuition of when a string appears complex or simple corresponds to the lengths of programs to print those strings. Simple strings should have short programs to print them.

2.1 Invariance of the Definition

Why did we say a program and not a Turing machine? It is like asymptotic analysis in the theory of algorithms. Our complexity measure is independent of the language it is written in. Unlike the theory of algorithms, rather than rely on our intuition, we can prove this independence.

Theorem 1. For any programming language l, Suppose we have a language specific function $K_l(x)$ which is the length of the shortest program in l to print x and halt. We prove that $K(x) \leq K_l(x)$ for some constant c. Choice of language can only matter by at most, a constant.

Proof. We prove that two language specific measures may differ from each other by at most a constant. Suppose we have language specific definitions for Python and Rust, denoted as $K_{\texttt{py}}$ and $K_{\texttt{rust}}$ respectively. By the Church-Turing Thesis, since Rust is Turing-complete, we can certainly write a Python interpreter in Rust. Let this program written in Rust to interpret Python be called $\pi_{\texttt{pyinrust}}$. Now, given any Python program, combined with this interpreter in Rust, we just have a Rust program. It might look something roughly like

```
fn interpret(python_code: &str) {
...
}
fn main() {
    let pyprog: &str = "#!/usr/bin/python3\ndef f()\n\t...";
    interpret(pyprog)
}
```

Suppose there is a python program *p.py* with $|p.py| = K_{\texttt{py}}(x)$. This minimal Python program can be used to create a Rust program, as seen above. We observe:

$$K_{\texttt{rust}}(x) \leq K_{\texttt{py}}(x) + |\pi_{\texttt{pyinrust}}| \tag{2}$$

We can only say $\leq$ and not $=$ since we do not know if there exists a smaller Rust program. But the existence of this Rust program which prints x upper bounds $K_{\texttt{rust}}(x)$. Notice, by the Church-Turing Thesis, that Python is also Turing-Complete. Thus, we can also write

a Rust interpreter in Python.[1] By a symmetrical argument, there exists a Rust interpreter in Python named $\pi_{\texttt{rustinpy}}$ and we observe:

$$K_{\texttt{py}}(x) \leq K_{\texttt{rust}}(x) + |\pi_{\texttt{rustinpy}}| \tag{3}$$

Notice that our interpreters are independent of anything about x, like its complexity or length. These interpreters are of constant size. You could have a python program of a billion gigabytes, and the code to interpret the python program would remain the same 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 there exists a constant c such that

$$\forall x \ |K_{\texttt{py}}(x) - K_{\texttt{rust}}(x)| \leq c \tag{4}$$

So, the difference between our two algorithmic complexities only differs by some constant.

This can obviously be generalized for all Turing-complete programming languages. We may drop the subscript and just consider $K(x)$ rightfully as a universal definition. □

Another small anecdote: This proof relied on a piece of evidence from the Church-Turing thesis. We took two Turing-complete programming languages and had them simulate each other.

3 Examples

Here are some examples to understand Kolmogorov complexity better. We can compute an upper bound on $K(x)$ for any x by simply giving a program to take no input and print x.

3.1 $K(x)$

Notice that for any $x \in \Sigma^*$, there exists a program to print it. Somewhat obviously, just have the program contain the string hardcoded. Such a program may look like.

```
def f():
    x = '........................'
    print(x)
```

This program takes no input and prints x, for any $x \in \Sigma^*$. So we observe that

$$\forall x \ K(x) \leq |x| + c \tag{5}$$

where c is some constant independent of the input. For example, |"`print()`"| = 7, so $c \geq 7$. It is independent of the string x, but dependent on the programming language, which we don't care about.

[1] A Rust interpreter in Python seems far less useful than a Python interpreter in Rust. Yet, you should imagine that someone could write such a program. For computability, we do not care about the difference between compiled and interpreted. A compiled language is really a translation into the language of machine instructions, which is then arguably just interpreted by the CPU. There do exist interpreters for traditionally compiled languages, like C. This distinction is unimportant. Arbitrary.

3.2 $K(xx)$

What about the Kolmogorov Complexity of some string concatenated with itself. What is $K(xx)$ in terms of x? A bad idea is to hardcode xx.

```
def badidea():
    xx = '............................................................'
    print(xx)
```

Rather than an upper bound of $K(xx) \leq |xx| + c = 2|x| + c$, we can construct a program to only store x instead of xx and then compute xx from x.

```
def goodidea():
    x = '........................'
    print(x.append(x))
```

This gives us a better upper bound of $K(xx) \leq |x| + c'$. Note that $c' > c$ since our program needs the logic to compute xx from x. It's still a constant though. For future reference, all constants are not necessarily equal, but all are independent of the input. Programs need not only store information, but they may compute it as well.

3.3 $K(x^n)$

What about many concatenations, like $K(x^n)$ for some n?

```
def f():
    x = '........................'
    n = .....
    ans = ''
    for i in range(n): ans.append(x)
    print(ans)
```

The size of our program as a function of the input is $|x|$ and $|n| = \log n$. So

$$K(x^n) \leq |x| + \log n + c \tag{6}$$

Conventionally, the size of a string x is its length $|x|$, but the size of a number n is the number of bits of its description, $\log n$. Before, we didn't need a $\log n$ term as it was only constantly many concatenations. Now we must keep a counter. Also notice we need not hardcode x. What if there was a much shorter way to compute x? Let g be some minimal program which takes no input and returns[2] x. Maybe its size is much smaller than the size of x ($|g| << |x|$).

```
def f():
    x = g()
    n = .....
```

[2]the difference between returning and printing is an engineering issue, and we don't care about the difference enough. Obviously if there is a program to return a string, there is a similarly sized program to print the string.

```
ans = ''
for i in range(n): ans.append(x)
print(ans)
```

Thus, $K(x^n) \leq K(x) + \log n + c$. We replaced the hardcoded x with a computation of x. We could even do the same for n if there is a shorter description than its hardcoded $\log n$ bits. This would give us an upper bound of $K(x^n) \leq K(x) + K(n) + c$.

3.4 $K(x^R)$

What about x^R, the reversal of string x? What is $K(x^R)$ in relation to $K(x)$? If some program p prints x, we can create a program q to print x^R. Note q is like p. It computes x, but instead of immediately printing it, it computes x, reverses it, then prints it. We observe this reversal operation is independent of the input, so $|q| = |p| + c$. Thus, $|K(x) - K(x^R)| \leq c$ for some constant c. This also underlines our intuition of K being a measure of natural descriptive complexity. If a string is complex or simple, its reversal should remain complex or simple. Our intuition on simple or complex strings is invariant to reversing.

4 Compression

Which strings have short descriptions and which ones have long ones? Let's get back to some intuition about randomness. Some strings appear to have very short, simple descriptions, like 1^{2^n}. A program to print this string needs to only really contain information about n, which is much smaller than the length of the string. Others strings appear to have long descriptions, or at least no short descriptions. We may say a string is incompressible if $K(x) \geq |x| - c$ for some c. The shortest description of an incompressible string isn't much shorter than the string itself. We may say a string is compressible if it is not incompressible. Which strings are compressible and which ones are incompressible?

Lets ask a simpler question. How many strings of length n are compressible by 2 or more bits? Just two bits. Let's compute it as a ratio:

$$\frac{\text{strings of length } n \text{ compressible by two bits}}{\text{all strings of length } n} = \frac{|\{x \in \Sigma^n \mid K(x) \leq |x| - 2\}|}{|\Sigma^n|} \leq \tag{7}$$

$$\frac{|\{p \in \Pi \mid \text{p a program with } |p| \leq n-2\}|}{2^n} \leq \frac{|\bigcup_{i=0}^{n-2} \Sigma^i|}{2^n} \leq \frac{\sum_{i=0}^{n-2} 2^i}{2^n} = \frac{2^{n-1}}{2^n} = \frac{1}{2} \tag{8}$$

Only half of the strings of length n are compressible by 2 bits. This generalizes so only $\frac{1}{4}$ are compressible by 3 bits and $\frac{1}{2^{d-1}}$ are compressible by d bits. This is a very lazy upper bound. It's much larger than actual amount. We were extremely generous with our overestimation and still concluded a very small upper bound. From this, we may conclude that overwhelmingly most strings are incompressible. Less than $\frac{1}{1000}$ strings are incompressible by 11 bits. Note this is independent of n.

The stress is on "most" strings. 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. If you had all possible files of size 100GB, only 0.1% of them could be compressed by more than a byte.

Why does file compression work in practice? Consider some fixed setting, like images of fixed dimension. By most we mean in a uniformly random sense, the color of each pixel being drawn according to a coin, you will generate an image which looks like garbage. In a uniformly random sense, "most" images look like TV static. In contrast, most of the useful images generated by humans are full of patterns for our pattern matching brain. A picture of a parrot may have a large splotch of red. Lossy encodings like JPEG and lossless algorithms like Lempel-Ziv exploit these patterns to generate short descriptions.

Back in the world of strings, the compressible strings are the lucky ones. If you were to generate the bits of a string by a random coin flip, its going to have overwhelming probability of having near equal number of zeroes and ones. With negligible, insignificant probability would it have any exploitable pattern or structure. How likely is the string xx or 1^n as an output of this part of this random process? Most strings are incompressible because most strings do not have any pattern.

Heres a deep remark. Although since we believe $\mathsf{P} \neq \mathsf{NP}$, we are unable to computationally distinguish random strings from those produced by a pseudo-random generator. Yet if an arbitrarily long random string is incompressible, arbitrarily long pseudo-random strings all have short descriptions. Those descriptions being simply the algorithm of the pseudo-random generator, the seed, and the string's length.

5 Graph of $K(x)$

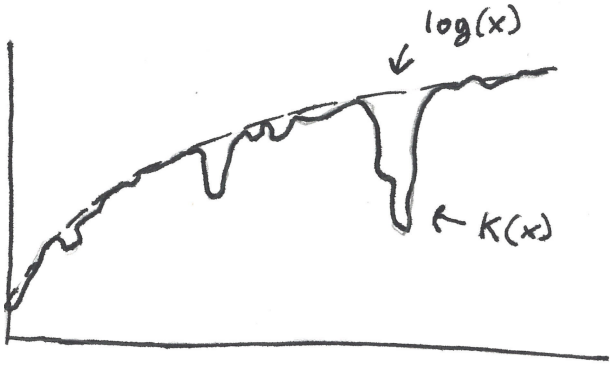
Lets try to plot $K(x)$, but instead of $K : \Sigma^* \to \mathbb{N}$, consider $K : \mathbb{N} \to \mathbb{N}$. We witness the following behavior of K.

- $K(x)$ grows unbounded. $\nexists c \ \forall x \ K(x) < c$. To prove this, consider n such that $K(1^n) > c$. Note that n can get really big, but c cannot. The function must be growing.
- $K(x)$ "hugs" $\log x$. We proved most strings are incrompressible, so the graph should hover near $\log x$ for most x.
- $K(x)$ dips infinitely often. A small program for one string implies an infinite family of small programs for an infinite family of strings. We showed that $K(x) \approx K(x^R)$. A description of a string being simple or complex does not depend on the direction we read it. This same intuition can be used to 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.
- $K(x)$ has continuous properties. Recall the definition of continuity of a real valued function. We say f is continuous if when $x, x+\varepsilon$ are close, so are $f(x), f(x+\varepsilon)$. This means $|x - x_0| < c_1 \implies |f(x) - f(x_0)| < c_2$. In terms of $K(x)$,

  $$\forall x, \ |K(x) - K(x \pm 1)| < c$$

  Take the program that prints x. Modify it to add 1, now you have a program to print $x+1$. Note that $K(x)$ cannot actually be continuous, as it is discretely valued, but it may not have a sporadic behavior

A plot of $K(x)$ with the following properties might look like the figure.



6 $K(x)$ is not computable

We have a imagined[3] graph of $K(x)$, but we never gave an algorithm. That's because there is none.

Theorem 2. $K(x)$ is not a computable function.

Proof. We will proceed by diagonalization. Assume to the contrary $K(x)$ is computable, and there is a program which may compute it. Then we may construct the following algorithm. This program is build around an assumed program to compute $K(x)$ It searches for the

Algorithm 1 M

```
for x ∈ Σ* lexographically do
    if K(x) > |⟨M⟩| then
        print x
        halt
    end if
end for
```

smallest lexographic string with Kolmogorov Complexity greater than the length of the code of the program. The for loop iterates like $x \in \{\varepsilon, 0, 1, 00, 01, \ldots\}$. As the algorithm proceeds, M will search for some smallest string x such that $K(x) > |\langle M \rangle|$ and print it. But since M itself prints this x, we see that $K(x) \leq |\langle M \rangle|$. A contradiction. It is impossible for both $a > b$ and $a \leq b$ to both be true. Thus, $K(x)$ is not computable. $\square$

The diagonalization occurs in the way we somehow were able to encode the size of a program within the program itself! This should surprise you. Diagonalization is a negated self-reference. The self reference occurs by encoding a program with its own size, and the negation occurs purely quantitatively. Can't draw a table here. If you were to write the rest of the program first, then compute $|\langle M \rangle|$, and try to insert it, you would change the size of the program. With a little math, this is avoidable. You could also replace $|\langle M \rangle|$ with a constant c which over estimates $|\langle M \rangle|$ significantly, and still derive a contradiction. There also exists something called Kleene's recursion theorem, which allows a program to obtain a copy of its own description, and compute with it.

[3] I traced this from the Li Vitanyi book

7 The Method of Incompressibility

This is a useful proof technique derived from Kolmogorov Complexity. Like how the pigeonhole principle shows existence of an object with some desired property, the method of incompressibility shows most objects have some desired property. Here, we mean "most", truly in a Kolmogorov-random sense. It is one of the strongest techniques we have for average case and worst case lower bounds. The proofs usually follow some similar structure. You assume something to the contrary, and then show that this implies some succinct description of an incompressible object.

7.1 Infinitude of the Primes

There is a classic proof due to Euclid you may know. There are many other proofs of this result as well. Here, we will prove it using the method of incompressibility.

Theorem 3. There are infinitely many primes.

Proof. Suppose there are only finitely many primes, $p_1, \ldots, p_k$. Then $\forall n \in \mathbb{N},\ \exists e_1, \ldots, e_k$ such that $n = p_1^{e_1} \cdots p_k^{e_k}$. Thus, the prime powers $\langle e_1, \ldots, e_k \rangle$ are a description of n. The program to print n would have hardcoded $e_1, ..., e_k$. It would bruteforce recompute all the primes, and then compute $n = {p_1}^{e_1} \cdot ... \cdot p_k^{e_k}$ and print it. Note that each e_i can be described in $\log e_i$ bits, so $K(n) \leq \log e_1 + \cdots + \log e_k + c$. What is the size of any e_i? A worst case is that n is a prime power, so if $n = p_i^{e_i}$ for some i, then $e_i = \log_{p_i} n$. It follows then that each $e_i \leq \log n$.

$$K(n) \leq \log e_1 + \cdots + \log e_k + c \leq \log\log n + \cdots \log\log n + c \leq k \log\log n + c \qquad (9)$$

where k is the finite number of primes and independent of the input. So,

$$\forall n,\ K(n) \leq k \log\log n + c$$

For any large enough incompressible n, we have a contradiction. □

We showed that if there were finitely many primes, then every number could be described succinctly by its prime powers. But we know most numbers are incompressible. We concluded that $\forall n,\ K(n) \leq k \log\log n$, but we know for most n that $K(n) > \log n - c$.

7.2 Proving non-regularity of languages

There is a generalized lemma in the Li-Vitanyi book, but here we only prove one example, as a demonstration.

Theorem 4. $\{a^n b^n \mid n \in \mathbb{N}\}$ is not regular.

Proof. Assume to the contrary that it was regular. Then there exists a DFA D to decide this language. Consider the execution of the machine on some input $a^n b^n$. Suppose you pause the execution right after the a's and before the b's. You will be paused on some (not accepting) state q_i. If you were to resume execution from this state q_i, after reading in exactly b^n, the DFA will be brought to an accept state. The first accept state that D

reaches from q_i will only come after reading b^n. We may use this fact to give a succinct description of n. Take the DFA D, run it from q_i on b's until you reach an accept state, and then print the number of b's it took. The program may look like

```
def f():
D = '.......................'
q = q_i
counter = 0
while True
    if q is accepting
        print counter and halt
    q = delta(q,b)
    counter+=1
print(ans)
```

We observe that $K(n) \leq |D| + |q_i| + c$. Since D, q_i are of constant size we see that $K(n) < c$, a contradiction for large enough incompressible n. □

Much easier than the pumping lemma! You can generalize this, like the pumping lemma, to give a generic proof template to show many non-regular languages are not regular.

8 A Hint Towards Computational Learning Theory

Suppose we loosened our definition of $K(x)$ so that the programs to print x need not be perfect, only approximate. Recall, our definition that

$$K(x) = \text{“the length of the shortest program to print string } x\text{”}$$

Suppose the following synonym substitutions were made:

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

Now, we have

$$K(x) = \text{“the size of the simplest description which approximates dataset } x\text{”}$$

That sounds a lot like Occam's Razor. Following this logic, you could formalize Occam's Razor under PAC[4] learning. In practice, since $K(x)$ is not computable, there is much more success with computable restrictions, such as

$$K^t(x) = \min_{p \in \Pi} \{|p| : U(p, \varepsilon) = x \text{ and halts in } t(|x|) \text{ steps}\} \tag{10}$$

[4]Probably Approximate Correct

9 Further Reading

You have to look at the Li Vitanyi book. Chapter two may guide you towards understanding more about the complexity itself. Chapter six will give you may applications of the method of incompressibility. These include Turing machine simulation lower bounds, average case complexity of heapsort, Hastad's switching lemma for circuit lower bounds, and much more. Chapter eight has some connections between Kolmogorov complexity and information theory. I also recommend you read 6.4 of the Sipser book and maybe this old worksheet of mine https://ladha.me/files/sectionX/kolmogorov.pdf