Information-Theoretic Modeling
Lecture 11: Further Topics

Jyrki Kivinen

Department of Computer Science, University of Helsinki

Autumn 2012
Lecture 11: Further Topics

(Peter Falk as *Columbo*, NBC)
1 Kolmogorov Complexity
   • Definition
   • Basic Properties
1. Kolmogorov Complexity
   - Definition
   - Basic Properties

2. Gambling
   - Gambler’s Ruin
   - Kelly Criterion
Kolmogorov Complexity

We probably agree that the string

\[ 101010101010101010101010101010 \ldots 10 \]

10 million characters

is “simple”.

Why?
Kolmogorov Complexity

We probably agree that the string

1010101010101010101010...10

10 million characters

is “simple”.

Why?

(One) Solution: The string has a short description:

“10 repeated 5 000 000 times”.
Kolmogorov Complexity

We probably agree that the string

\[ 10101010101010101010\ldots10 \]

is “simple”.

Why?

(One) Solution: The string has a short description:

“10 repeated 5 000 000 times”.

Remark: “Description” should be understood to mean a code that can be decoded by some algorithm (a formal procedure that halts).
To precisely define Kolmogorov complexity we need to fix a formal notion of algorithm.
To precisely define Kolmogorov complexity we need to fix a formal notion of algorithm.

Following tradition, we use here Turing Machine (TM), but any other universal model of computation could be used as well. For simplicity, we assume that the inputs and outputs are strings over the binary alphabet \( \{0, 1\} \).
To precisely define Kolmogorov complexity we need to fix a formal notion of algorithm.

Following tradition, we use here Turing Machine (TM), but any other universal model of computation could be used as well. For simplicity, we assume that the inputs and outputs are strings over the binary alphabet \( \{0, 1\} \).

If TM \( U \) on input \( p \in \{0, 1\}^* \) halts and outputs \( x \in \{0, 1\}^* \), we write \( U(p) = x \).

If \( U \) does not halt on input \( p \), we say that \( U(p) \) is undefined and write \( U(p) = \emptyset \).

We use \( |p| \) to denote the length of string \( x \).
The **Kolmogorov complexity** of string $x \in \{0, 1\}^*$ with respect to a Turing machine $U$ is defined as the length of shortest input on which $U$ outputs $x$:

$$K_U(x) = \min \{|p| \mid U(p) = x \}.$$
Kolmogorov Complexity

The **Kolmogorov complexity** of string $x \in \{0, 1\}^*$ with respect to a Turing machine $U$ is defined as the length of shortest input on which $U$ outputs $x$:

$$K_U(x) = \min \{ |p| \mid U(p) = x \}.$$

Notice that $K_U(x)$ depends on our choice of $U$, which at this stage is arbitrary. However, we can remove most of this dependence by limiting the set of machines $U$ we consider.
Kolmogorov Complexity

The **Kolmogorov complexity** of string $x \in \{0, 1\}^*$ with respect to a Turing machine $U$ is defined as the length of shortest input on which $U$ outputs $x$:

$$K_U(x) = \min \{ |p| \mid U(p) = x \}.$$  

Notice that $K_U(x)$ depends on our choice of $U$, which at this stage is arbitrary. However, we can remove most of this dependence by limiting the set of machines $U$ we consider.

- Turing machine $U$ is a **prefix machine** if the set of inputs on which $U$ halts is prefix-free.
The **Kolmogorov complexity** of string \( x \in \{0, 1\}^* \) with respect to a Turing machine \( U \) is defined as the length of shortest input on which \( U \) outputs \( x \):

\[
K_U(x) = \min \{ |p| \mid U(p) = x \}.
\]

Notice that \( K_U(x) \) depends on our choice of \( U \), which at this stage is arbitrary. However, we can remove most of this dependence by limiting the set of machines \( U \) we consider.

- Turing machine \( U \) is a **prefix machine** if the set of inputs on which \( U \) halts is prefix-free.
- Turing machine \( U \) is **universal** if for any other TM \( V \) there is a string \( q_V \) such that \( V(p) = U(q_V p) \) for all \( p \in \{0, 1\}^* \).
The **Kolmogorov complexity** of string \( x \in \{ 0, 1 \}^* \) with respect to a Turing machine \( U \) is defined as the length of shortest input on which \( U \) outputs \( x \):

\[
K_U(x) = \min \{ |p| \mid U(p) = x \}
\]

Notice that \( K_U(x) \) depends on our choice of \( U \), which at this stage is arbitrary. However, we can remove most of this dependence by limiting the set of machines \( U \) we consider.

- Turing machine \( U \) is a **prefix machine** if the set of inputs on which \( U \) halts is prefix-free.
- Turing machine \( U \) is **universal** if for any other TM \( V \) there is a string \( q_V \) such that \( V(p) = U(q_V p) \) for all \( p \in \{ 0, 1 \}^* \).

Here \( qp \) means concatenation of strings \( q \) and \( p \).
Prefix property in practice

Requiring prefix property may seem a bit technical, but intuitively it just means we must be able to tell when the input ends.
Prefix property in practice

Requiring prefix property may seem a bit technical, but intuitively it just means we must be able to tell when the input ends.

In practical programming, this is done by methods such as end-of-file markers, or having the operating system keep track of file lengths.
Prefix property in practice

Requiring prefix property may seem a bit technical, but intuitively it just means we must be able to tell when the input ends.

In practical programming, this is done by methods such as end-of-file markers, or having the operating system keep track of file lengths.

End-of-file markers actually are a way of keeping the input set prefix free. However reserving one symbol for this special use is not generally acceptable if we are interested in optimal code lengths.
Prefix property in practice

Requiring prefix property may seem a bit technical, but intuitively it just means we must be able to tell when the input ends.

In practical programming, this is done by methods such as end-of-file markers, or having the operating system keep track of file lengths.

End-of-file markers actually are a way of keeping the input set prefix free. However reserving one symbol for this special use is not generally acceptable if we are interested in optimal code lengths.

Theoretically more satisfying way to make a set of inputs prefix-free is to include the length of (the rest of) the input string using some prefix code.
A straightforward prefix code for integers is the following:

Consider integer $x$ with $n$ bit binary representation $x_1x_2x_3 \ldots x_n$.
We encode $x$ as $x_1x_1x_2x_2x_3x_3 \ldots x_nx_n01$.
We denote this code for $x$ by $\langle x \rangle$. 


A straightforward prefix code for integers is the following:

Consider integer $x$ with $n$ bit binary representation $x_1x_2x_3\ldots x_n$. We encode $x$ as $x_1x_1x_2x_2x_3x_3\ldots x_nx_n01$. We denote this code for $x$ by $\langle x \rangle$.

For example, for $x = 19 = 10011_2$ we get $\langle x \rangle = 110000111101$. 
Prefix code for integers

A straightforward prefix code for integers is the following:

Consider integer \( x \) with \( n \) bit binary representation \( x_1 x_2 x_3 \ldots x_n \). We encode \( x \) as \( x_1 x_1 x_2 x_2 x_3 x_3 \ldots x_n x_n 01 \).

We denote this code for \( x \) by \( \langle x \rangle \).

For example, for \( x = 19 = 10011_2 \) we get \( \langle x \rangle = 110000111101 \).

The length of the prefix-free encoding is \( |\langle x \rangle| = 2n + 2 \) bits, where \( n = \lceil \log_2(x + 1) \rceil \leq \log_2 x + 1 \) is the length of the original binary representation.
Prefix property for Turing Machines

We can now make the set of inputs prefix-free by inserting before each input $x$ its length encoded as $\langle |x| \rangle$. For example, input 1000100101101, which has 13 bits, becomes 1111001101. More generally, an input of $n$ bits gets code length of at most $n + 2 \log_2 n + 2$ bits. For large $n$ this is much better than $2^n + 2$ we would get by applying the prefix-free encoding from previous slide directly to $x$. To summarize, the prefix property is reasonable from a practical point of view, and from a theoretical point of view can be assumed without increasing input lengths too much.
Prefix property for Turing Machines

We can now make the set of inputs prefix-free by inserting before each input $x$ its length encoded as $\langle |x| \rangle$.

For example, input 1000100101101, which has 13 bits, becomes

$$\underbrace{1111001101}_{\text{code for 13 = 1101}_2} \underbrace{1000100101101}_{\text{actual input}}.$$
Prefix property for Turing Machines

We can now make the set of inputs prefix-free by inserting before each input $x$ its length encoded as $\langle |x| \rangle$.

For example, input 1000100101101, which has 13 bits, becomes

$$\underbrace{1111001101}_{\text{code for } 13 = 1101_2}\underbrace{1000100101101}_{\text{actual input}}.$$ 

More generally, an input of $n$ bits gets code length of at most $n + 2 \log_2 n + 2$ bits.
Prefix property for Turing Machines

We can now make the set of inputs prefix-free by inserting before each input $x$ its length encoded as $\langle |x| \rangle$.

For example, input 1000100101101, which has 13 bits, becomes

$$1111001101 1000100101101$$

code for 13 = $1101_2$ actual input

More generally, an input of $n$ bits gets code length of at most $n + 2 \log_2 n + 2$ bits.

For large $n$ this is much better than $2n + 2$ we would get by applying the prefix-free encoding from previous slide directly to $x$. 
Prefix property for Turing Machines

We can now make the set of inputs prefix-free by inserting before each input $x$ its length encoded as $\langle |x| \rangle$.

For example, input 1000100101101, which has 13 bits, becomes

$$1111001101 \quad 1000100101101$$

code for $13 = 1101_2$ actual input.

More generally, an input of $n$ bits gets code length of at most $n + 2 \log_2 n + 2$ bits.

For large $n$ this is much better than $2n + 2$ we would get by applying the prefix-free encoding from previous slide directly to $x$.

To summarize, the prefix property is reasonable from a practical point of view, and from a theoretical point of view can be assumed without increasing input lengths too much.
Universal Turing Machines

For any fixed $x$, there are Turing machines that output $x$ with empty input, and other Turing machines that don’t output $x$ with any input.
For any fixed $x$, there are Turing machines that output $x$ with empty input, and other Turing machines that don’t output $x$ with any input.

Similarly, for any pair of different strings $x \neq y$, there are Turing machines $U$ and $V$ such that $K_U(x) \ll K_U(y)$ but $K_V(x) \gg K_V(y)$. 
Universal Turing Machines

For any fixed $x$, there are Turing machines that output $x$ with empty input, and other Turing machines that don’t output $x$ with any input.

Similarly, for any pair of different strings $x \neq y$, there are Turing machines $U$ and $V$ such that $K_U(x) \ll K_U(y)$ but $K_V(x) \gg K_V(y)$.

For comparisons of Kolmogorov complexities to be meaningful, we require $U$ to be universal.
Universal Turing Machines

Universality

A Turing Machine $U$ is said to be **universal**, if for any other Turing Machine $V$ there is a string $q_V \in \{0, 1\}^*$ (which depends on $V$) such that for all strings $p$ we have

$$U(q_V p) = V(p).$$

That is when given the concatenated input $qp$, TM $U$ outputs the same string as TM $V$ when given input $p$. 
Universal Turing Machines

Universality

A Turing Machine $U$ is said to be **universal**, if for any other Turing Machine $V$ there is a string $q_V \in \{0, 1\}^*$ (which depends on $V$) such that for all strings $p$ we have

$$U(q_V p) = V(p).$$

That is when given the concatenated input $qp$, TM $U$ outputs the same string as TM $V$ when given input $p$.

If we think of strings $p$ as programs in the “machine language” of $V$, then $q_V$ is an “interpreter” or “compiler” for $V$’s machine language, written for machine $U$ (in the machine language of $U$).
Examples of (virtually) universal ‘computers’:
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
Examples

Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
3. your favorite programming language (compiler/interpreter + OS + computer),
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
3. your favorite programming language (compiler/interpreter + OS + computer),
4. Universal Turing machine,
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
3. your favorite programming language (compiler/interpreter + OS + computer),
4. Universal Turing machine,
5. Universal recursive function,
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
3. your favorite programming language (compiler/interpreter + OS + computer),
4. Universal Turing machine,
5. Universal recursive function,
6. Lambda calculus,
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
3. your favorite programming language (compiler/interpreter + OS + computer),
4. Universal Turing machine,
5. Universal recursive function,
6. Lambda calculus,
7. Arithmetics,
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
3. your favorite programming language (compiler/interpreter + OS + computer),
4. Universal Turing machine,
5. Universal recursive function,
6. Lambda calculus,
7. Arithmetics,
8. Game of Life
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
3. your favorite programming language (compiler/interpreter + OS + computer),
4. Universal Turing machine,
5. Universal recursive function,
6. Lambda calculus,
7. Arithmetics,
8. Game of Life
9. ...
Examples of (virtually) universal ‘computers’:

1. C (compiler + operating system + computer),
2. Java (compiler + operating system + computer),
3. your favorite programming language (compiler/interpreter + OS + computer),
4. Universal Turing machine,
5. Universal recursive function,
6. Lambda calculus,
7. Arithmetics,
8. Game of Life
9. ...

Each of the above can mimic all the others.
Kolmogorov Complexity

For any *universal* computer \( U \), and any other computer \( V \), we have

\[
K_U(x) \leq K_V(x) + C,
\]

where \( C \) is a constant independent of \( x \).
For any universal computer $U$, and any other computer $V$, we have

$$K_U(x) \leq K_V(x) + C,$$

where $C$ is a constant independent of $x$.

**Proof:** Let $q_V$ be such that $U(q_V p) = V(p)$ for all $p$. Let $p_V^*(x)$ be the shortest program for which $V(p_V^*(x)) = x$. Then $U(q_V p_V^*(x)) = x$, so

$$K_U(x) \leq |q_V p_V^*(x)| = |p_V^*(x)| + |q_V| = K_V(x) + |q_V|.$$

□
Kolmogorov Complexity

For any *universal* computer $U$, and any other computer $V$, we have

$$K_U(x) \leq K_V(x) + C,$$

where $C$ is a constant independent of $x$.

**Proof:** Let $q_V$ be such that $U(q_V p) = V(p)$ for all $p$. Let $p^*_V(x)$ be the shortest program for which $V(p^*_V(x)) = x$. Then $U(q_V p^*_V(x)) = x$, so

$$K_U(x) \leq |q_V p^*_V(x)| = |p^*_V(x)| + |q_V| = K_V(x) + |q_V|.$$

Since we are restricting ourselves to prefix machines, we don’t need to worry about any overhead caused by encoding the pair $(q_V, p)$, we can just concatenate them.
Invariance Theorem

From now on we restrict the choice of the computer $U$ in $K_U$ to universal computers.
Invariance Theorem

From now on we restrict the choice of the computer $U$ in $K_U$ to universal computers.

**Invariance Theorem**

Kolmogorov complexity is invariant (up to an additive constant) under a change of the universal computer. In other words, for any two universal computers, $U$ and $V$, there is a constant $C$ such that

$$|K_U(x) - K_V(x)| \leq C \text{ for all } x \in \{0, 1\}^*. $$
Invariance Theorem

From now on we restrict the choice of the computer $U$ in $K_U$ to universal computers.

**Invariance Theorem**

Kolmogorov complexity is invariant (up to an additive constant) under a change of the universal computer. In other words, for any two universal computers, $U$ and $V$, there is a constant $C$ such that

$$|K_U(x) - K_V(x)| \leq C \quad \text{for all } x \in \{0, 1\}^*.$$  

**Proof:** Since $U$ is universal, we have $K_U(x) \leq K_V(x) + C_1$. Since $V$ is universal, we have $K_V(x) \leq K_U(x) + C_2$. The theorem follows by setting $C = \max\{C_1, C_2\}$. 

$\square$
Kolmogorov Complexity

Upper Bound

We have the following upper bound on $K_U(x)$:

$$K_U(x) \leq |x| + 2 \log_2 |x| + C$$

for some constant $C$ which depends on the computer $U$ but not on the string $x$. 
Upper Bound

We have the following upper bound on $K_U(x)$:

$$K_U(x) \leq |x| + 2 \log_2 |x| + C$$

for some constant $C$ which depends on the computer $U$ but not on the string $x$.

**Proof:** Remember that we have a prefix code where the code length for $x$ is

$$\ell(x) = |x| + 2 \log_2 |x| + 2.$$

Let $V$ be a TM that decodes this encoding. Then $K_V(x) = \ell(x)$. Therefore, for universal $U$ we have $K_U(x) \leq \ell(x) + C$. \qed
Conditional Kolmogorov Complexity

The **conditional Kolmogorov complexity** is defined as the length of the shortest program to print $x$ when $y$ is given:

$$K_U(x \mid y) = \min \{ |p| \mid U(\bar{y} \ p) = x \},$$

where $\bar{y}$ is a prefix-encoded representation of $y$. 

**Upper Bound**

We have the following upper bound on $K_U(x \mid x)$:

$$K_U(x \mid x) \leq |x| + C$$

for some constant $C$ independent of $x$. 

Jyrki Kivinen

Information-Theoretic Modeling
Conditional Kolmogorov Complexity

The **conditional Kolmogorov complexity** is defined as the length of the shortest program to print $x$ when $y$ is given:

$$K_U(x \ | \ y) = \min \{ |p| \ | U(\bar{y} \ p) = x \} ,$$

where $\bar{y}$ is a prefix-encoded representation of $y$.

Upper Bound 2

We have the following upper bound on $K_U(x \ | \ |x|)$:

$$K_U(x \ | \ |x|) \leq |x| + C$$

for some constant $C$ independent of $x$. 
Examples

Let $n = |x|$. 
Examples

Let $n = |x|$.

1. $K_U(0101010101\ldots01 \mid n) = C$.

   **Program:** print $n/2$ times 01.
Let \( n = |x| \).

1. \( K_U(0101010101\ldots01 \mid n) = C \).
   
   \textit{Program:} print \( n/2 \) times 01.

2. \( K_U(\pi_1 \pi_2 \ldots \pi_n \mid n) = C \).
   
   \textit{Program:} print the first \( n \) bits of \( \pi \).
Examples

Let \( n = |x| \).

1. \( K_U(0101010101\ldots01 \mid n) = C. \)
   
   **Program:** print \( n/2 \) times 01.

2. \( K_U(\pi_1 \pi_2 \ldots \pi_n \mid n) = C. \)
   
   **Program:** print the first \( n \) bits of \( \pi \).

3. \( K_U(\text{English text} \mid n) \approx 1.3 \times n + C. \)
   
   **Program:** Huffman code.
   (Entropy of English is about 1.3 bits per symbol.)
Examples

Let $n = |x|$.

1. $K_U(0101010101\ldots01 \mid n) = C$.
   
   **Program:** print $n/2$ times 01.

2. $K_U(\pi_1\pi_2\ldots\pi_n \mid n) = C$.
   
   **Program:** print the first $n$ bits of $\pi$.

3. $K_U(\text{English text} \mid n) \approx 1.3 \times n + C$.
   
   **Program:** Huffman code.
   
   (Entropy of English is about 1.3 bits per symbol.)

4. $K_U(\text{fractal}) = C$.
   
   **Program:** print \# of iterations until $z_{n+1} = z_n^2 + c > T$.
Examples
Examples (contd.):

5. \( K_U(x \mid n) \approx n \), for almost all \( x \in \{0, 1\}^n \).
Martin-Löf Randomness

Examples (contd.):

5. $K_U(x | n) \approx n$, for almost all $x \in \{0, 1\}^n$.

Proof: Upper bound $K_U(x | n) \leq n + C$. Lower bound by a counting argument: less than $2^{-k}$ of strings compressible by more than $k$ bits (Lecture 1).
Examples (contd.):

5. \( K_U(x \mid n) \approx n \), for almost all \( x \in \{0, 1\}^n \).

**Proof:** Upper bound \( K_U(x \mid n) \leq n + C \). Lower bound by a counting argument: less than \( 2^{-k} \) of strings compressible by more than \( k \) bits (Lecture 1).

**Martin-Löf Randomness**

String \( x \) is said to be **Martin-Löf random** iff \( K_u(x \mid n) \geq n \).
Examples (contd.):

5. $K_U(x \mid n) \approx n$, for almost all $x \in \{0, 1\}^n$.

**Proof:** Upper bound $K_U(x \mid n) \leq n + C$. Lower bound by a counting argument: less than $2^{-k}$ of strings compressible by more than $k$ bits (Lecture 1).

**Martin-Löf Randomness**

String $x$ is said to be **Martin-Löf random** iff $K_u(x \mid n) \geq n$.

Consequence of point 5 above: An i.i.d. sequence of unbiased coin flips is with high probability Martin-Löf random.
Universal Prediction

Since the set of valid (halting) programs is required to be **prefix-free** we can consider the probability distribution \( p^n_U \):

\[
p^n_U(x) = \frac{2^{-K_U(x|n)}}{C}, \quad \text{where } C = \sum_{x \in \mathcal{X}^n} 2^{-K_U(x|n)}.
\]
Universal Prediction

Since the set of valid (halting) programs is required to be **prefix-free** we can consider the probability distribution $p^n_U$:

$$p^n_U(x) = \frac{2^{-K_U(x|n)}}{C}, \quad \text{where} \quad C = \sum_{x \in \mathcal{X}^n} 2^{-K_U(x|n)}.$$

**Universal Probability Distribution**

The distribution $p^n_U$ is universal in the sense that for any other computable distribution $q$, there is a constant $C > 0$ such that

$$p^n_U(x) \geq C \cdot q(x) \quad \text{for all} \quad x \in \mathcal{X}^n.$$
Universal Prediction

Since the set of valid (halting) programs is required to be **prefix-free** we can consider the probability distribution \( p^n_U \):

\[
p^n_U(x) = \frac{2^{-K_U(x|n)}}{C}, \quad \text{where } C = \sum_{x \in \mathcal{X}^n} 2^{-K_U(x|n)}.
\]

### Universal Probability Distribution

The distribution \( p^n_U \) is universal in the sense that for any other computable distribution \( q \), there is a constant \( C > 0 \) such that

\[
p^n_U(x) \geq C \ q(x) \quad \text{for all } x \in \mathcal{X}^n.
\]

**Proof idea:** The universal computer \( U \) can imitate the Shannon-Fano prefix code with codelengths \( \left\lceil \log_2 \frac{1}{q(x)} \right\rceil \).
Universal Prediction

The universal probability distribution $p^n_U$ is a good predictor.
Universal Prediction

The universal probability distribution $p^n_U$ is a good predictor.

This follows from the relationship between codelengths and probabilities (Kraft!):

$$K_U(x) \text{ is small } \Rightarrow p^n_U(x) \text{ is large}$$
Universal Prediction

The universal probability distribution $p^n_U$ is a good predictor.

This follows from the relationship between codelengths and probabilities (Kraft!):

$$K_U(x) \text{ is small } \Rightarrow p^n_U(x) \text{ is large}$$

$$\Rightarrow \prod_{i=1}^{n} p^n_U(x_i \mid x_1, \ldots, x_{i-1}) \text{ is large}$$
Universal Prediction

The universal probability distribution $p^n_U$ is a good predictor.

This follows from the relationship between codelengths and probabilities (Kraft!):

$$K_U(x) \text{ is small } \Rightarrow p^n_U(x) \text{ is large}$$

$$\Rightarrow \prod_{i=1}^{n} p^n_U(x_i \mid x_1, \ldots, x_{i-1}) \text{ is large}$$

$$\Rightarrow p^n_U(x_i \mid x_1, \ldots, x_{i-1}) \text{ is large for most } i \in \{1, \ldots, n\},$$

where $x_i$ denotes the $i$th bit in string $x$. 
Berry Paradox

The smallest integer that cannot be described in ten words?

Whatever this number is, we have just described it in ten words.

Whatever this number is, it is quite interesting!
Berry Paradox

The smallest integer that cannot be described in ten words?

Whatever this number is, we have just described (?) it in ten words.
Berry Paradox

The smallest integer that cannot be described in ten words?

Whatever this number is, we have just described (?) it in ten words.

The smallest uninteresting number?
Berry Paradox

The smallest integer that cannot be described in ten words?

Whatever this number is, we have just described (?) it in ten words.

The smallest uninteresting number?

Whatever this number is, it is quite interesting!
Non-computability

It is impossible to construct a general procedure (algorithm) to compute $K_U(x)$.

Non-Computability

Kolmogorov complexity $K_U : \{0, 1\}^* \rightarrow \mathbb{N}$ is non-computable.
Non-computability

It is impossible to construct a general procedure (algorithm) to compute $K_U(x)$.

Non-Computability

Kolmogorov complexity $K_U : \{0, 1\}^* \rightarrow \mathbb{N}$ is non-computable.

Proof: Assume, by way of contradiction, that it would be possible to compute $K_U(x)$. 
Non-computability

It is impossible to construct a general procedure (algorithm) to compute \( K_U(x) \).

**Non-Computability**

Kolmogorov complexity \( K_U : \{0, 1\}^* \rightarrow \mathbb{N} \) is **non-computable**.

**Proof:** Assume, by way of contradiction, that it would be possible to compute \( K_U(x) \). Then for any \( M > 0 \), the program

print a string \( x \) for which \( K_U(x) > M \).

would print a string with \( K_U(x) > M \).
Non-computability

It is impossible to construct a general procedure (algorithm) to compute $K_U(x)$.

Non-Computability

Kolmogorov complexity $K_U : \{0, 1\}^* \rightarrow \mathbb{N}$ is non-computable.

**Proof:** Assume, by way of contradiction, that it would be possible to compute $K_U(x)$. Then for any $M > 0$, the program

print a string $x$ for which $K_U(x) > M$.

would print a string with $K_U(x) > M$. A contradiction follows by letting $M$ be larger than the Kolmogorov complexity of this program. Hence, it cannot be possible to compute $K_U(x)$.  

Jyrki Kivinen  Information-Theoretic Modeling
Universal codes (or models) in MDL were defined to be universal with respect to some model class $\mathcal{M}$, which from an application point of view is “user-specified”.
Universal codes (or models) in MDL were defined to be universal \emph{with respect to} some model class $\mathcal{M}$, which from an application point of view is “user-specified”.

There are universal codes with respect to quite general model classes (such as Lempel-Ziv for finite-order Markov models), but still this may feel a bit unsatisfactory from a philosophical point of view.
Universal codes (or models) in MDL were defined to be universal with respect to some model class $\mathcal{M}$, which from an application point of view is “user-specified”.

There are universal codes with respect to quite general model classes (such as Lempel-Ziv for finite-order Markov models), but still this may feel a bit unsatisfactory from a philosophical point of view.

Kolmogorov complexity gives a code that is universal with respect to any computable model class, which seems the best we can hope for.
Unfortunately Kolmogorov complexity itself is not computable, limiting its applicability in practice. However Kolmogorov complexity is useful as an idealization and for understanding our limitations.
Summary: Kolmogorov complexity and MDL

Unfortunately Kolmogorov complexity itself is not computable, limiting its applicability in practice. However Kolmogorov complexity is useful as an idealization and for understanding our limitations.

One should also remember that even in principle, Kolmogorov complexity is defined only up to an additive constant (depending on the choice of $U$).
1. Kolmogorov Complexity
   - Definition
   - Basic Properties

2. Gambling
   - Gambler’s Ruin
   - Kelly Criterion
Gambling

Winning strategies for horse racing at any track!

Betting on Horse Racing FOR DUMMIES

A Reference for the Rest of Us!

Richard Eng

Can't figure out the odds? Want to make money? This book shows you how.

Expected win $\mathbb{E}[b] = \sum p_x \alpha_x b_x$.

Maximized by betting everything on $\text{arg max} p_x \alpha_x$. 

Jyrki Kivinen

Information-Theoretic Modeling
Bet money $b_x$ on horse $x$. Get money $\alpha_x b_x$ if $x$ wins (odds).
Bet money $b_x$ on horse $x$. Get money $\alpha_x b_x$ if $x$ wins (odds).

Expected win $E[b_x \alpha_x] = \sum p_x \alpha_x b_x$. 
Bet money $b_x$ on horse $x$. Get money $\alpha_x b_x$ if $x$ wins (odds).

Expected win $E[b_x \alpha_x] = \sum p_x \alpha_x b_x$.

Maximized by betting everything on $\arg \max p_x \alpha_x$. 
If odds are “fair”, then \( \alpha_x = \frac{1}{p_x} \), and hence \( p_x \alpha_x b_x = b_x \) for all \( i \).
If odds are "fair", then $\alpha_x = \frac{1}{p_x}$, and hence $p_x \alpha_x b_x = b_x$ for all $i$.

Assume now that we have **inside information** about the winning horse.
If odds are “fair”, then $\alpha_x = \frac{1}{p_x}$, and hence $p_x \alpha_x b_x = b_x$ for all $i$.

Assume now that we have inside information about the winning horse.

In the extreme case, $\hat{X} = X$, we know the outcome:

$$V_n = \alpha_{x_1} \alpha_{x_2} \cdots \alpha_{x_n} V_0$$

where $V_t$ is the capital on $t$th step.
Gambling

If odds are “fair”, then $\alpha_x = \frac{1}{p_x}$, and hence $p_x \alpha_x b_x = b_x$ for all $i$.

Assume now that we have inside information about the winning horse.

In the extreme case, $\hat{X} = X$, we know the outcome:

$$V_n = \alpha_{x_1} \alpha_{x_2} \cdots \alpha_{x_n} V_0 = \left(2^G\right)^n V_0$$

where $V_t$ is the capital on $t$th step, and $G = \frac{\log \sum \alpha_{x_i}}{n}$.
Gambling

If odds are “fair”, then $\alpha_x = \frac{1}{p_x}$, and hence $p_x \alpha_x b_x = b_x$ for all $i$.

Assume now that we have **inside information** about the winning horse.

In the extreme case, $\hat{X} = X$, we know the outcome:

$$V_n = \alpha_{x_1} \alpha_{x_2} \cdots \alpha_{x_n} \ V_0 = \left(2^G\right)^n \ V_0$$

where $V_t$ is the capital on $t$th step, and $G = \log \sum_{n} \alpha_{x_i}$.
If the channel is noisy, so that \( q_{x_i} = p(x_i | \hat{x}_i) < 1 \), then our final capital is

\[
V_n = \alpha_{x_1} \beta_{x_1|\hat{x}_1} \alpha_{x_2} \beta_{x_2|\hat{x}_2} \cdots \alpha_{x_n} \beta_{x_n|\hat{x}_n} V_0,
\]

where \( \beta_{x_i|\hat{x}_i} = \frac{b_{x_i}}{V_{i-1}} \) is the proportion of capital on \( x_i \) given \( \hat{x}_i \).
If the channel is noisy, so that $q_{x_i} = p(x_i \mid \hat{x}_i) < 1$, then our final capital is

$$V_n = \alpha_{x_1} \beta_{x_1 \mid \hat{x}_1} \alpha_{x_2} \beta_{x_2 \mid \hat{x}_2} \cdots \alpha_{x_n} \beta_{x_n \mid \hat{x}_n} V_0,$$

where $\beta_{x_i \mid \hat{x}_i} = \frac{b_{x_i}}{V_{i-1}}$ is the proportion of capital on $x_i$ given $\hat{x}_i$.

Again, expected wealth maximized by betting everything on

$$\arg \max q_{x_i} \alpha_{x_i}.$$
If the channel is noisy, so that $q_{x_i} = p(x_i \mid \hat{x}_i) < 1$, then our final capital is

$$V_n = \alpha_{x_1} \beta_{x_1 \mid \hat{x}_1} \alpha_{x_2} \beta_{x_2 \mid \hat{x}_2} \cdots \alpha_{x_n} \beta_{x_n \mid \hat{x}_n} V_0,$$

where $\beta_{x_i \mid \hat{x}_i} = \frac{b_{x_i}}{V_{i-1}}$ is the proportion of capital on $x_i$ given $\hat{x}_i$.

Again, expected wealth maximized by betting everything on $\arg \max q_{x_i} \alpha_{x_i}$.

**Gambler’s Ruin**

This strategy is guaranteed to lead to bankruptcy sooner or later!
If the channel is noisy, so that \( q_{x_i} = p(x_i \mid \hat{x}_i) < 1 \), then our final capital is

\[
V_n = \alpha x_1 \beta_{x_1 \mid \hat{x}_1} \alpha x_2 \beta_{x_2 \mid \hat{x}_2} \cdots \alpha x_n \beta_{x_n \mid \hat{x}_n} V_0,
\]

where \( \beta_{x_i \mid \hat{x}_i} = \frac{b_{x_i}}{V_{i-1}} \) is the proportion of capital on \( x_i \) given \( \hat{x}_i \).

Again, expected wealth maximized by betting everything on \( \arg \max q_{x_i} \alpha_{x_i} \).

**Gambler’s Ruin**

This strategy is guaranteed to lead to bankruptcy sooner or later!

**Conclusion:** Maximum expected wealth is not the thing to consider.
What if we maximize the average growth rate of capital instead?

\[ G = \frac{1}{n} \log \frac{V_n}{V_0} = \frac{1}{n} \log \prod_{i=1}^{n} \alpha_{x_i} \beta_{x_i | \hat{x}_i}. \]
What if we maximize the average growth rate of capital instead?

\[
G = \frac{1}{n} \log \frac{V_n}{V_0} = \frac{1}{n} \sum_{i=1}^{n} \log \alpha_{x_i} \beta_{x_i} | \hat{x}_i.
\]
Maximum Growth Rate

What if we maximize the average growth rate of capital instead?

\[ G = \frac{1}{n} \log \frac{V_n}{V_0} = \frac{1}{n} \sum_{i=1}^{n} \log \alpha_{x_i} \beta_{x_i} | \hat{x}_i . \]

With fair odds \( \alpha_{x_i} = \frac{1}{p_{x_i}} \), this becomes

\[ G = \frac{1}{n} \sum_{i=1}^{n} \log \frac{\beta_{x_i} | \hat{x}_i}{p_{x_i}} \]
Maximum Growth Rate

What if we maximize the average growth rate of capital instead?

\[ G = \frac{1}{n} \log \frac{V_n}{V_0} = \frac{1}{n} \sum_{i=1}^{n} \log \alpha_{x_i} \beta_{x_i} | \hat{x}_i. \]

With fair odds \( \alpha_{x_i} = \frac{1}{p_{x_i}} \), this becomes

\[ E[G] = \frac{1}{n} \sum_{i=1}^{n} E \left[ \log \frac{\beta_{x_i} | \hat{x}_i}{p_{x_i}} \right] \]
What if we maximize the average growth rate of capital instead?

\[ G = \frac{1}{n} \log \frac{V_n}{V_0} = \frac{1}{n} \sum_{i=1}^{n} \log \alpha_{x_i} \beta_{x_i | \hat{x}_i}. \]

With fair odds \( \alpha_{x_i} = \frac{1}{p_{x_i}} \), this becomes

\[ E[G] = \frac{1}{n} \sum_{i=1}^{n} E \left[ \log \frac{\beta_{x_i | \hat{x}_i}}{p_{x_i}} \right] = \sum_{x, \hat{x} \in \mathcal{X}} p_{x, \hat{x}} \log \frac{\beta_{x_i | \hat{x}_i}}{p_x} \]
Maximum Growth Rate

What if we maximize the average growth rate of capital instead?

\[ G = \frac{1}{n} \log \frac{V_n}{V_0} = \frac{1}{n} \sum_{i=1}^{n} \log \alpha_{x_i} \beta_{x_i|x_i}. \]

With fair odds \( \alpha_{x_i} = \frac{1}{p_{x_i}} \), this becomes

\[ E[G] = \frac{1}{n} \sum_{i=1}^{n} E \left[ \log \frac{\beta_{x_i|x_i}}{p_{x_i}} \right] = \sum_{x, \hat{x} \in \mathcal{X}} p_{x, \hat{x}} \log \frac{\beta_{x_i}}{p_{x}} \]

\[ + \sum_{\hat{x} \in \mathcal{X}} p_{\hat{x}} \sum_{x \in \mathcal{X}} p_{x|\hat{x}} \log \beta_{x_i|\hat{x}_i} + H_p(X) \]
Maximum Growth Rate

What if we maximize the average growth rate of capital instead?

\[
G = \frac{1}{n} \log \frac{V_n}{V_0} = \frac{1}{n} \sum_{i=1}^{n} \log \alpha_{x_i} \beta_{x_i | \hat{x}_i}.
\]

With fair odds \( \alpha_{x_i} = \frac{1}{p_{x_i}} \), this becomes

\[
E[G] = \frac{1}{n} \sum_{i=1}^{n} E \left[ \log \frac{\beta_{x_i | \hat{x}_i}}{p_{x_i}} \right] = \sum_{x, \hat{x} \in \mathcal{X}} p_{x, \hat{x}} \log \frac{\beta_{x_i | \hat{x}_i}}{p_{x}} + H_p(X).
\]

Gibbs’ inequality: Maximized by \( \beta_{x_i | \hat{x}_i} = q_{x_i} = p_{x_i | \hat{x}_i} \).
Theorem (Kelly, 1956)

Assuming fair odds, \( \alpha_x = \frac{1}{p_x} \),

the growth rate \( G \) is maximized by betting proportion \( q_x = p(x \mid \hat{x}) \) of the capital on \( x \in \mathcal{X} \),
Kelly Criterion

Theorem (Kelly, 1956)

Assuming fair odds, $\alpha_x = \frac{1}{p_x}$,

1. the growth rate $G$ is maximized by betting proportion $q_x = p(x \mid \hat{x})$ of the capital on $x \in \mathcal{X}$,

2. then the growth rate is given by

$$G = H(X) - H(X \mid \hat{X}),$$

i.e., the channel capacity,
Theorem (Kelly, 1956)

Assuming fair odds, $\alpha_x = \frac{1}{p_x}$,

1. the growth rate $G$ is maximized by betting proportion $q_x = p(x \mid \hat{x})$ of the capital on $x \in \mathcal{X}$,

2. then the growth rate is given by

$$G = H(X) - H(X \mid \hat{X}),$$

i.e., the channel capacity,

3. gambling using any other strategy will eventually yield less profit.
The same strategy is optimal even if the odds are not fair in the sense $\alpha_x = \frac{1}{p_x}$, as long as there is no “track take”, i.e.,

$$\sum_{x \in X} \frac{1}{\alpha_x} = 1.$$
The same strategy is optimal even if the odds are not fair in the sense $\alpha_x = \frac{1}{p_x}$, as long as there is no “track take”, i.e.,

$$\sum_{x \in X} \frac{1}{\alpha_x} = 1.$$ 

Note that this implies that you should ignore the odds when betting!
The same strategy is optimal even if the odds are not fair in the sense \( \alpha_x = \frac{1}{p_x} \), as long as there is no “track take”, i.e.,

\[
\sum_{x \in X} \frac{1}{\alpha_x} = 1.
\]

Note that this implies that you should ignore the odds when betting!

The analysis can be extended to the case where there is a “track take”, but the results are not quite as neat.
The End.