Data compression, character encoding

Coding Technology

Illés Horváth

2025/10/22

Source coding

We have a source that generates a sequence of characters from a given finite set $\mathcal{X}$ called source alphabet (of size N).

During source coding, we want to assign a bit sequence (codeword) to each character:

$$X_i \to c_i, \qquad i = 1, \ldots, N.$$

The most natural character encoding is to assign a codeword of the same length to each character. This is called fixed length encoding.

When the probabilities of each character are close to each other, this is a reasonable choice, but when the probabilities are very different, variable-length encoding may be better.

Character encoding

Let's construct a variable-length code! (Keep in mind that we also want to be able to decode fully.)

Alphabet: {A, B, C, D}.

A	0
B	110
C	1111
D	0

Character encoding

Let's construct a variable-length code! (Keep in mind that we also want to be able to decode fully.)

Alphabet: {A, B, C, D}.

A	0
B	110
C	1111
D	1

Character encoding

Let's construct a variable-length code! (Keep in mind that we also want to be able to decode fully.)

Alphabet: {A, B, C, D}.

A	0
B	110
C	1111
D	11

Character encoding

Let's construct a variable-length code! (Keep in mind that we also want to be able to decode fully.)

Alphabet: {A, B, C, D}.

A	0
B	110
C	1111
D	10

Coding Technology

General setup.

During the Coding Technology course, our main focus is on various types of codes:

- Error correction codes adding redundancy to the messages that allow detection and/or correction of errors, allowing reliable communication over noisy channels. Also known as Error Control, or Channel Coding.
- **Data compression codes** identify patterns and eliminate redundancies in data. Also known as Source Coding.
- Cryptography protocols that turn readable information into unintelligible text, allowing secure private communication over public channels.

Data compression

Source coding generally refers to the process of encoding information by a bit sequence as short as possible.

The term data compression means generally the same as source coding, but it is often used when the original information is also represented as a bit sequence.

There are two main branches of data compression:

- in lossless data compression, we want to be able to decode the original information perfectly;
- in lossy data compression, unnecessary or less important parts of the original information may be lost.

We will start with lossless methods.

Memoryless stationary source

An *information source* is an object or process that generates information that we would like to communicate.

We will use the following model for the information source, called *memoryless stationary source*:

- the source generates a sequence of characters from a given finite set $\mathcal{X}$ called *source alphabet*;
- the characters are random and independent from each other (memoryless);
- the probability of each character remains constant over the sequence (stationary).

We assume that the probability distribution of the source alphabet is known.

Memoryless stationary source

Examples (source alphabet):

- a sequence of dice rolls ($\{1, 2, 3, 4, 5, 6\}$)
- computer files (bytes)
- English text (English alphabet)
- DNA sequence ($\{G, A, T, C\}$)
- sheet music (notes)

Are the assumptions justified? That depends on the source. . .

A fixed alphabet and stationarity are usually justified.

The memoryless property might not be justified; it's rather a simplifying assumption/approximation ensuring we do not have to worry about long-term structures in the source, only characters.

(But we will look for long-term structures later.)

The memoryless property is sometimes replaced by weaker assumptions (e.g. Markov source).

Character encoding

Character encoding is when we assign a bit sequence (codeword) to each character of the source alphabet:

$$f : X_i \to c_i, \qquad i = 1, \ldots, N$$

(where N denotes the size of the source alphabet $\mathcal{X}$.)

There are other types of source coding; for example, we could code several characters together (block coding). We will also explore this later.

When using character encoding, the code corresponding to a string (multiple characters) is coded as the concatenation of the codewords.

We say that a character encoding is *uniquely decodable* if, for any finite two distinct character sequences $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$,

$$f(x_1) \ldots f(x_n) \neq f(y_1) \ldots f(y_m)$$

(where $f(x_1)f(x_2)$ denotes concatenation).

Character encoding

A character encoding is called *fixed length encoding* if it assigns a codeword of the same length to each character. All fixed length encodings are uniquely decodable.

We will refer to fixed length codeword assignments as 'no compression'. For a source alphabet of size N, the necessary length is $\lceil \log_2 N \rceil$.

When the probabilities of each character are close to each other, this is a reasonable choice, but when the probabilities are very different, *variable-length encoding* may be better.

Character encoding

A character encoding is called a *prefix code* (also called prefix-free code) if none of the codewords is a prefix (initial segment) of another codeword.

Example. Our original code is a prefix code:

A	0
B	110
C	1111
D	10

Actually, all fixed length encodings are also prefix codes.

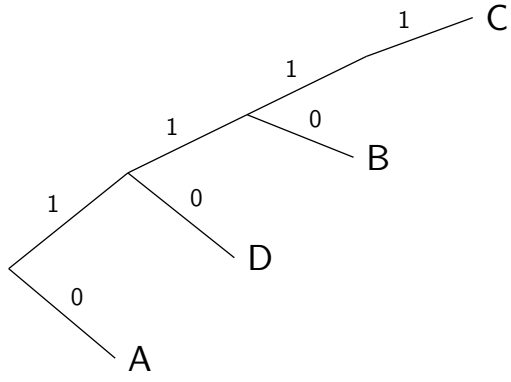
Character encoding

Theorem

Any prefix code is uniquely decodable.

Proof. Prefix codes can be decoded easily using a code tree:

A	0
B	110
C	1111
D	10



For prefix codes, codewords appear only in leaves (nodes at the edge of the tree); we decode a bit sequence by starting from the current position and progressing along the tree until we reach a codeword. Then one character is decoded, and we restart from the next position and the root of the tree. □

Kraft–McMillan inequality

Are prefix codes the only uniquely decodable codes?

No; for example prefix codes written backwards (suffix codes) are also uniquely decodable.

But no other uniquely decodable codes are better.

Theorem (Kraft–McMillan)

(a) *For any uniquely decodable code with codeword lengths* $\ell_1, \ldots, \ell_N$,

$$\sum_{i=1}^{N} 2^{-\ell_i} \leq 1.$$

(b) *For any prescribed codeword lengths* $\ell_1, \ldots, \ell_N$ *such that*

$$\sum_{i=1}^{N} 2^{-\ell_i} \leq 1,$$

there exists a prefix code with codeword lengths $\ell_1, \ldots, \ell_N$.

Kraft–McMillan inequality

Proof.

(a) Let k be a positive integer, and let A_ℓ denote the number of bit sequences of length ℓ that can be obtained by concatenating k codewords.

Since the code is uniquely decodable, $A_\ell \leq 2^\ell$.

Let $L_{\max} := \max(\ell_1, \ldots, \ell_N)$.

$$\left(\sum_{i=1}^{N} 2^{-\ell_i}\right)^k = \sum_{i_1=1}^{k} \cdots \sum_{i_N=1}^{k} 2^{-(\ell_{i_1}+\cdots+\ell_{i_k})} = \sum_{\ell=1}^{kL_{\max}} A_\ell 2^{-\ell} \leq kL_{\max}.$$

But then, taking $\sqrt[k]{\ }$,

$$\sum_{i=1}^{N} 2^{-\ell_i} \leq \underbrace{\sqrt[k]{k}}_{\to 1} \underbrace{\sqrt[k]{L_{\max}}}_{\to 1} \to 1 \quad \text{as } k \to \infty,$$

so $\sum_{i=1}^{N} 2^{-\ell_i} \leq 1$ must hold.

Kraft–McMillan inequality

Proof.

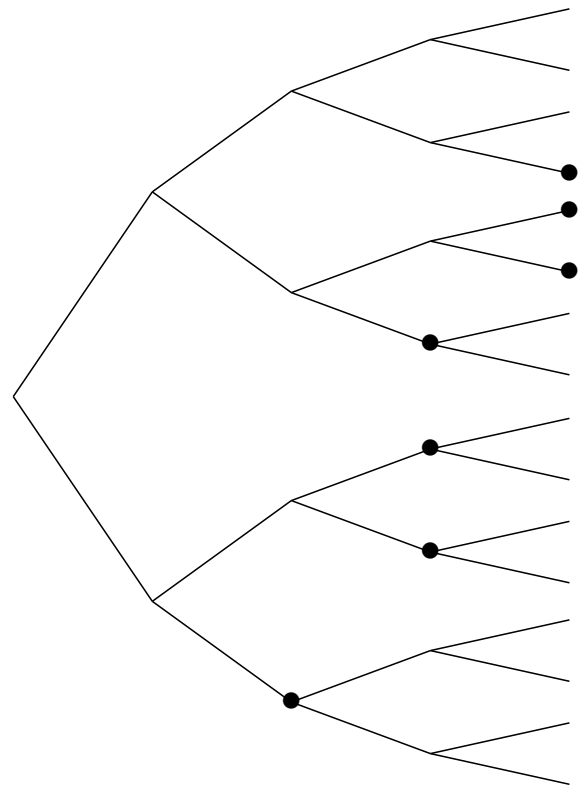
(b) Assume the prescribed codeword lengths are in increasing order:

$$\ell_1 \leq \cdots \leq \ell_N$$

Start from a full binary tree of depth $L_{\max}(= \ell_N)$, and going from bottom to top in the tree, for each ℓ_i in increasing order, set a node at depth ℓ_i, truncating deeper nodes.

Example: $2, 3, 3, 3, 4, 4, 4$.

Finally, we truncate unused branches.



Kraft–McMillan inequality

(b) (cont.) Do we ever get stuck with the previous algorithm?

Calculate the number of nodes at depth $L_{\max}$ 'covered' by each codeword. A codeword of length ℓ_i covers $2^{L_{\max}-\ell_i}$ nodes at depth $L_{\max}$. Altogether, they cover

$$\sum_{i=1}^{N} 2^{L_{\max}-\ell_i} = 2^{L_{\max}} \sum_{i=1}^{N} 2^{-\ell_i} \leq 2^{L_{\max}} \cdot 1$$

nodes at depth $L_{\max}$ according to the assumption, and there are $2^{L_{\max}}$ nodes at depth $L_{\max}$, so there is enough room to accommodate all nodes corresponding to the prescribed codeword lengths, and the algorithm will not get stuck before finishing.

It is also important that we go in increasing order of codeword length, so each node covers an entire branch of the tree. □

Average codeword length

The typical way to measure the efficiency of a character encoding is via the average codeword length

$$L = \sum_{i=1}^{N} p_i \ell_i$$

Generally, we want L to be as low as possible.

Example. Assume we have a source with source alphabet {A,B,C,D} and character probabilities

$$p_A = 0.40, \quad p_B = 0.32, \quad p_C = 0.18, \quad p_D = 0.10.$$

Average codeword length

$$p_A = 0.40, \quad p_B = 0.32, \quad p_C = 0.18, \quad p_D = 0.10$$

The coding

A	0
B	110
C	1111
D	10

gives average codeword length

$$L = 0.40 \cdot 1 + 0.32 \cdot 3 + 0.18 \cdot 4 + 0.10 \cdot 2 = 2.28.$$

Shannon–Fano code

How to construct an efficient prefix code (or, equivalently, the corresponding code tree) in general, assuming the character probability distribution $p_1, \ldots, p_N$ is known?

One possible approach is the following. Order the characters in decreasing order of probability: $p_1 \geq \cdots \geq p_N$.

For the character X_i, the codeword length is going to be

$$\ell_i = \lceil -\log_2 p_i \rceil.$$

Then

$$\sum_{i=1}^{N} 2^{-\ell_i} = \sum_{i=1}^{N} 2^{-\lceil -\log_2 p_i \rceil} \leq \sum_{i=1}^{N} 2^{\log_2 p_i} = \sum_{i=1}^{N} p_i = 1,$$

so the Kraft–McMillan construction can be applied.

This is known as the Shannon–Fano code.

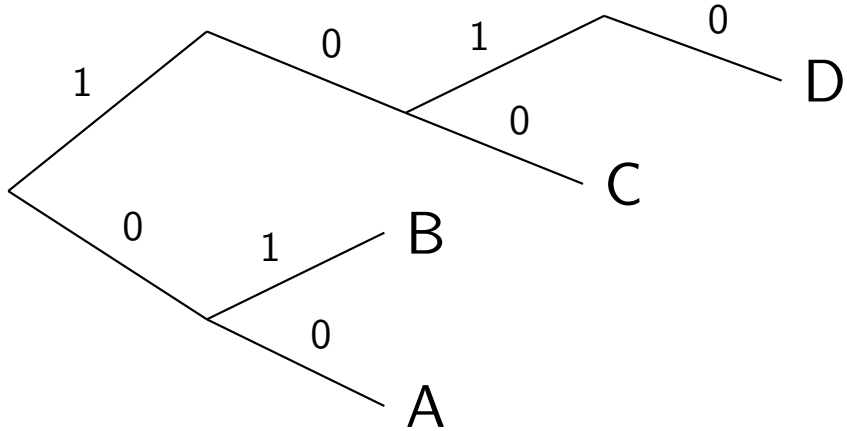
Shannon–Fano code

Example. For the previous example

$$p_A = 0.40, \quad p_B = 0.32, \quad p_C = 0.18, \quad p_D = 0.10,$$

the codeword lengths according to the Shannon–Fano code are

$$\ell_1 = \lceil -\log_2 0.40 \rceil = 2, \quad \ell_2 = \lceil -\log_2 0.32 \rceil = 2,$$
$$\ell_3 = \lceil -\log_2 0.18 \rceil = 3, \quad \ell_4 = \lceil -\log_2 0.10 \rceil = 4,$$



A	00
B	01
C	100
D	1010

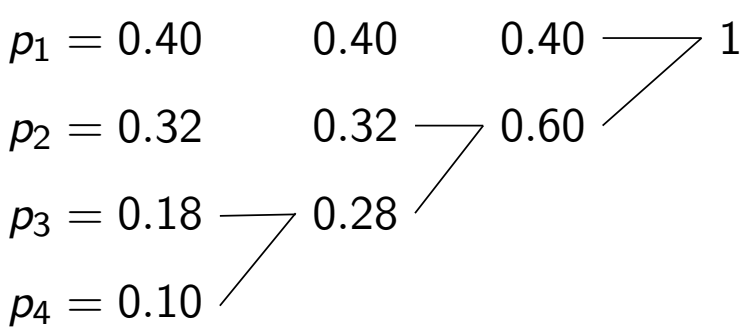
and the average codeword length is

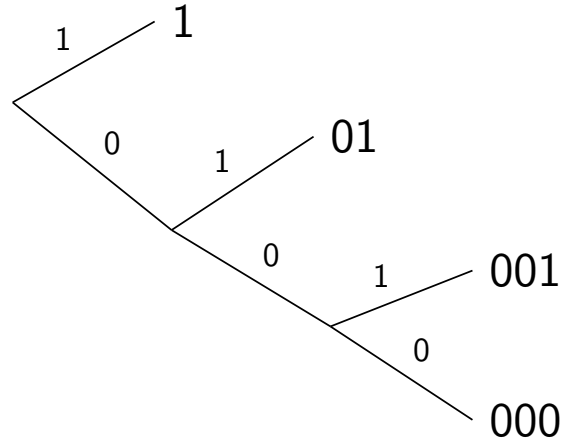
$$L = 0.40 \cdot 2 + 0.32 \cdot 2 + 0.18 \cdot 3 + 0.10 \cdot 4 = 2.44.$$

Huffman coding

Huffman coding builds the coding tree by adding the two smallest p_i probabilities in each step.

Example. $p_1 = 0.40, \quad p_2 = 0.32, \quad p_3 = 0.18, \quad p_4 = 0.10.$





The average codeword length is

$$L = 0.40 \cdot 1 + 0.32 \cdot 2 + 0.18 \cdot 3 + 0.10 \cdot 3 = 1.88.$$

Entropy

Huffman code gave the best average codeword length L for the running example so far. But how low can we go anyway?

The *entropy* of a source is

$$H(X) = \sum_{i=1}^{N} p_i \log_2(1/p_i).$$

The main theoretical result for today: entropy is a theoretical lower bound on the average codeword length L.

Theorem (Shannon's source coding theorem)

For any prefix code with codeword lengths ℓ_i $(i = 1, \ldots, N)$,

$$L \geq H(X).$$

Entropy

Before proving the theorem, we address entropy.

How to measure information?

Usually we measure information in bits; for example, learning the value of a fair coin flip is 1 bit of information.

We consider two things:

- The information learned from a random event depends on the probability of that event. Generally, a lower probability event occurring contains more information.
- The probability of independent random events is multiplicative, but we want to measure information in an additive manner.

Both of these issues are solved if we take $-\log_2$ of the probabilities (base 2 is just for convenience).

Entropy

(The $-$ sign is included since probabilities are between 0 and 1, so $\log_2$ of a probability is negative.)

For a memoryless stationary source, learning that the next character is X_i corresponds to

$$-\log_2 p_i = \log_2(1/p_i)$$

bits of information.

With all that in mind, the entropy

$$H(X) = \sum_{i=1}^{N} p_i \log_2(1/p_i)$$

is the average amount of information obtained by learning one character of the source.

Convexity, Jensen's inequality

To prove Shannon's source coding theorem, we need some mathematical tools.

We say that $f : [a, b] \to \mathbb{R}$ is convex if $\forall x, y \in [a, b]$ and $\forall 0 < \lambda < 1$,

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y).$$

Theorem (Jensen's inequality)

Let Z be a random variable that takes values from $[a, b]$, and f be a convex function on $[a, b]$. Then

$$f(E(Z)) \leq E(f(Z)).$$

No proof.

Gibbs' inequality

The following result is a consequence of Jensen's inequality.

Lemma (Gibbs' inequality)

If $p_1, \ldots, p_N > 0$, $q_1, \ldots, q_N > 0$, *and*

$$\sum_{i=1}^{N} p_i = \sum_{i=1}^{N} q_i = 1,$$

then

$$-\sum_{i=1}^{N} p_i \log p_i \leq -\sum_{i=1}^{N} p_i \log q_i$$

Proof (sketch). Apply Jensen's inequality to the random variable

$$P\left(Z = \frac{q_i}{p_i}\right) = p_i, \quad i = 1, \ldots, N$$

and the convex function $f(x) = -\log_2 x$, then rearrange. □

Shannon's source coding theorem

Proof (Shannon's source coding theorem).

Assume we have a prefix code with codeword lengths $\ell_1, \ldots, \ell_N$. Apply the previous Lemma with

$$p_i = p(X_i), \qquad q_i = \frac{2^{-\ell_i}}{\sum_{i=1}^{N} 2^{-\ell_i}}$$

Then

$$H(X) = -\sum_{i=1}^{N} p_i \log_2 p_i \leq -\sum_{i=1}^{N} p_i \log_2 \left(\frac{2^{-\ell_i}}{\sum_{i=1}^{N} 2^{-\ell_i}} \right) =$$

$$-\sum_{i=1}^{N} p_i \underbrace{\log_2(2^{-\ell_i})}_{-\ell_i} + \log_2 \underbrace{\sum_{i=1}^{N} 2^{-\ell_i}}_{\leq 1 \text{ due to Kraft–McMillan}} \leq L + \log_2 1 = L$$

Shannon's source coding theorem

There are other versions of Shannon's source coding theorem that apply not only to character encodings, but any other types of encodings that are uniquely decodable. We do not pursue this direction (so 'no proof').

The main takeaway from Shannon's source coding theorem is that a source cannot be compressed into a number of bits fewer than the amount of information (as defined by entropy) if we want it to be fully decodable.

Moreover, Shannon's source coding theorem also applies not only to memoryless stationary sources, but other types of sources too (although entropy may have a slightly different definition, depending on the type of the source).

Entropy

For the running example

$$p_1 = 0.40, \quad p_2 = 0.32, \quad p_3 = 0.18, \quad p_4 = 0.10,$$

the entropy of the source is

$$\begin{aligned} H(X) = {} & 0.40 \cdot \log_2(1/0.40) + 0.32 \cdot \log_2(1/0.32) \\ & + 0.18 \cdot \log_2(1/0.18) + 0.10 \cdot \log_2(1/0.10) \approx 1.832. \end{aligned}$$

Reminder: for this source,

$$\begin{aligned} L_{\text{Huff}} &= 1.88, \\ L_{\text{S–F}} &= 2.44, \\ L_{\text{fixed}} &= 2. \end{aligned}$$

Entropy

For any prefix code, $L \geq H(X)$; the *efficiency* of the code is the ratio $H(X)/L$.

Lemma

$$H(X) \leq L_{S\text{–}F} \leq H(X) + 1.$$

Proof. Directly follows from

$$-\log_2 p_i \leq \lceil -\log_2 p_i \rceil \leq -\log_2 p_i + 1,$$

averaged out according to the distribution p_i $(i = 1, \ldots, N)$. □

So the Shannon–Fano code gives average codeword length L within a distance of 1 bit from the theoretical lower bound.

Huffman is optimal

Theorem

Huffman coding gives the minimal average codeword length L from among all uniquely decodable character encodings.

Proof (sketch). Assume $p_1 \geq \cdots \geq p_N > 0$ for simplicity. Due to Kraft–McMillan, it is sufficient to prove Huffman is optimal among prefix codes.

For any optimal coding with codewords c_i $(i = 1, \ldots, N)$ and codeword lengths ℓ_i $(i = 1, \ldots, N)$, we may assume the following properties:

- $\ell_1 \leq \cdots \leq \ell_N$, otherwise we could rearrange the codewords;
- $\ell_{N-1} = \ell_N$, otherwise we could truncate c_N to length ℓ_{N-1}, and
- c_{N-1} and c_N only differ in the last bit, otherwise we could replace c_N by c_{N-1} with the last bit changed.

Huffman is optimal

Proof (cont.) We do mathematical induction on the number of characters in the source alphabet N.

Assume we have an optimal prefix coding for N source characters, with c_N and c_{N-1} only differing in the last bit. Let c'_{N-1} be their common section of length $\ell_{N-1} - 1$, and consider the coding

$$c_1, \ldots, c_{N-2}, c'_{N-1}$$

for a source alphabet of $N - 1$ characters with probabilities $p_1, \ldots, p_{N-2}, p_{N-1} + p_N$.

This coding on $N - 1$ characters must be optimal, otherwise we could take an optimal coding on $N - 1$ characters and split the longest codeword into two codewords by adding a 0 and a 1 at the end, obtaining a coding on N characters that would be better than the original, which was assumed optimal.

But this merging of the two smallest probabilities is exactly what Huffman coding does. □

Huffman is optimal

Lemma

For a given character probability distribution $p_1, \ldots, p_N$, Huffman coding gives

$$L = H(X) \quad \Longleftrightarrow \quad \log_2(1/p_i) \in \mathbb{Z}^+ \quad \forall i \in \{1, \ldots, N\}$$

Proof (sketch). During the construction of the Huffman code, when merging the two smallest probabilities, those two smallest probabilities have to be equal, otherwise we lose $L = H(X)$.

This must be true in every step of the Huffman algorithm in order to have $L = H(X)$, which is possible if and only if every p_i is a negative integer power of 2. □

Entropy coding

So it turns out Huffman coding reaches the theoretical lower bound for only some source probability distributions.

Huffman coding is optimal among character encodings, so this also means that for some sources, the theoretical lower bound cannot be reached with any character encoding.

That said, entropy can still be reached in this cases as well – just not with character encodings, but more complicated codings instead.

Arithmetic coding

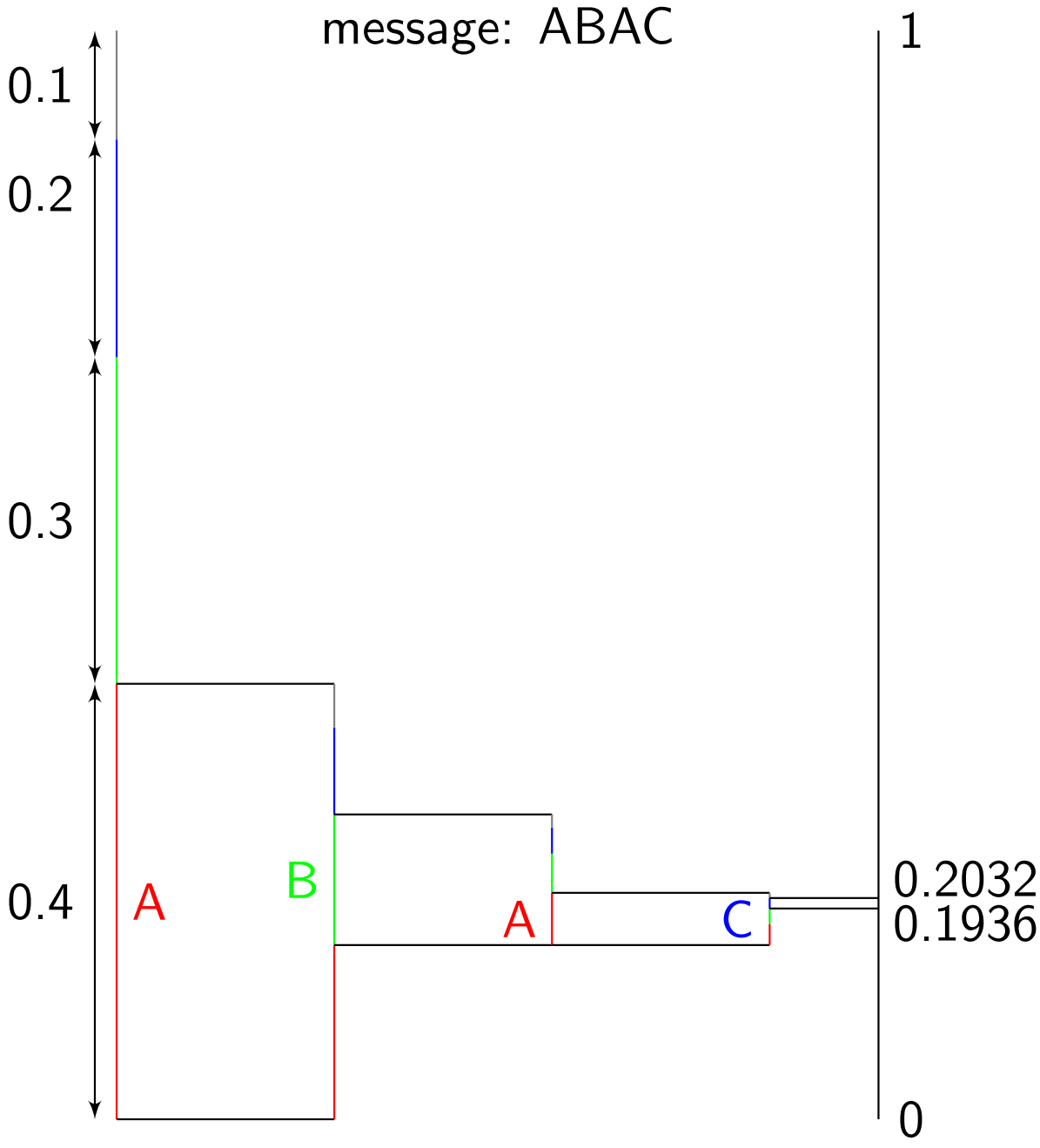
Shannon–Fano coding was inefficient because the $\lceil . \rceil$ function was applied to each character separately. Arithmetic coding (AC) is based on the same idea, but instead of coding characters separately, AC compresses the entire message at once.

Example. The alphabet is {A,B,C,D}, with

$$P(A) = 0.40, \quad P(B) = 0.30, \quad P(C) = 0.20, \quad P(D) = 0.10.$$

For AC, the compressed message will correspond to a single point from $[0, 1)$, obtained via consecutive sub-intervals.

Arithmetic coding – example



Arithmetic coding – example

The interval corresponding to the message ABAC is $[0.1936, 0.2032]$.

We want to use the middle point of this interval (in binary form) as the compressed message:

$$0.1984 = 0.00110010110\ldots_{②}$$

The main question: how many bits of precision do we need so we can distinguish this interval from the other small intervals?

Arithmetic coding – example

The number of bits required is

$$\lceil -\log_2(P(A)P(B)P(A)P(C)) \rceil + 1 = 8,$$

because then the rounding error is smaller than $P(A)P(B)P(A)P(C)/2$, so even the rounded value will be inside the same interval:

$$\begin{aligned} 0.1936 &= 0.00110001100\ldots \\ 0.1984 &\approx 0.00110011 \\ 0.2032 &= 0.00110100000\ldots \end{aligned}$$

Arithmetic coding

AC is not a character encoding, so it can be better than Huffman coding. In fact, for long messages, the compression rate will asymptotically converge to the entropy lower bound: for a character sequence $C_1 \dots C_n$,

$$\lim_{n\to\infty} \frac{1}{n}\left(\left\lceil -\log_2\left(\prod_{i=1}^{n} P(C_i)\right)\right\rceil + 1\right) = \lim_{n\to\infty} -\frac{1}{n}\log_2\left(\prod_{i=1}^{n} P(C_i)\right)$$

$$= \lim_{n\to\infty} -\frac{1}{n}\sum_{i=1}^{n}\log_2 P(C_i) = \sum_{k=1}^{K} p_k \log_2(1/p_k) = H(X)$$

due to the Law of Large Numbers.

AC can be decoded online: decoding can be started using the beginning of the compressed message, with more and more of the message decoded as further sections of the compressed message are received.

Computational complexity, data structures

We didn't address computational complexity of the above codings so far, but all of Huffman, Shannon–Fano and Arithmetic coding have relatively low computational complexity for both coding and decoding. (And of course fixed length encodings too.)

Huffman and Shannon–Fano use binary trees (for the code tree).

A binary tree is a data structure where each record contains some information stored in the node, plus pointers to the parent and the two (or fewer) children of the corresponding node.