Stream codes

Last modified: 2026-11-21 00:32:35 (PST)

A Huffman code gives each value a whole number of bits, so its expected length can exceed the entropy by nearly one bit per value. This page shows how to code a whole message at once, so that the waste is shared among all of its values.

Arithmetic coding

Definition 0.1 (Arithmetic code)**.** Let X be a discrete random variable whose values come in a fixed order, with probability mass function $\mathbb{p}$, and let $F(x) = s_{x'<x}\mathbb{p}(x')$ for the values x' before x. For a message $x = (x_1, \dots, x_n)$ of n independent copies of X, the **arithmetic code** is built as follows:

1. Start with the slice $[0, 1)$.
2. For $i = 1, \dots, n$: if the current slice is $[a, b)$, replace it with the part of $[a, b)$ that x_i owns, which is $[a + (b-a)F(x_i),\ a + (b-a)(F(x_i) + \mathbb{p}(x_i)))$.
3. Let $[a, a+w)$ be the final slice, and let $m = a + w/2$ be its midpoint. The codeword $c(x)$ is the first $\ell(x)$ bits of m after the point, where the length $\ell(x) = \lceil -\log_2 w \rceil + 1$ is part of this definition.

Example 0.1 (An arithmetic code for a short message)**.**
Take a source with the values a, b and c, in that order, and the probabilities $(0.5, 0.3, 0.2)$. The code below follows Definition 0.1 for the message a, c, b, a. It prints the slice after each round, then the codeword and its codeword length.

```
p <- c(a = 0.5, b = 0.3, c = 0.2)
cdf_before <- cumsum(p) - p
message_values <- c("a", "c", "b", "a")
n <- length(message_values)
lower <- 0
width <- 1
lowers <- numeric(n)
widths <- numeric(n)
for (i in seq_len(n)) {
  symbol <- message_values[i]
  lower <- lower + width * cdf_before[[symbol]]
  width <- width * p[[symbol]]
  lowers[i] <- lower
  widths[i] <- width
}
uppers <- lowers + widths
data.frame(
  symbol = message_values,
  lower = lowers,
  upper = uppers,
  width = widths
)
#>   symbol lower upper width
#> 1      a  0.00 0.500 0.500
#> 2      c  0.40 0.500 0.100
#> 3      b  0.45 0.480 0.030
#> 4      a  0.45 0.465 0.015
midpoint <- lower + width / 2
surprisal <- -log2(width)
n_digits <- ceiling(surprisal) + 1
positions <- seq_len(n_digits)
digits <- floor(midpoint * 2^positions) %% 2
codeword <- paste0(digits, collapse = "")
c(midpoint = midpoint, codeword_length = n_digits, surprisal = surprisal)
#>        midpoint codeword_length       surprisal
#>        0.457500        8.000000        6.058894
codeword
#> [1] "01110101"
```

The final slice has width 0.015, which is the product $0.5 \times 0.2 \times 0.3 \times 0.5$ of the probabilities of the four values. The codeword is 01110101, with 8 digits, which is the message's

surprisal of 6.059 bits rounded up to a whole number, plus one.

Lemma 0.1 (The intervals of an arithmetic code)**.** *In the arithmetic code for n independent copies of X, the final slice of a message $x = (x_1, \dots, x_n)$ has width $\mathbb{p}(x_1) \cdots \mathbb{p}(x_n)$, and the final slices of two different messages do not overlap.*

Proof

Proof. Each round replaces a slice of width $b - a$ by a slice of width $(b - a)\mathbb{p}(x_i)$. After n rounds the width is $\mathbb{p}(x_1) \cdots \mathbb{p}(x_n)$.
The parts that the values of X own are the slices $[F(x), F(x) + \mathbb{p}(x))$, which do not overlap, because each starts where the one before it ends. So the parts of one slice do not overlap either.
Let x and x' be two different messages, and let j be the first position where they differ. Before round j, both messages have the same slice. At round j they take different parts of it, which do not overlap. Later rounds keep each slice inside its own part, so the final slices do not overlap. □

Theorem 0.1 (Arithmetic codes are prefix codes)**.** *Let X be a discrete random variable whose values each have probability. Then the codewords of the arithmetic code for n independent copies of X form a prefix code on the messages of length n.*

Proof

Proof. Let x be a message, let $[a, a + w)$ be its final slice, and let $m = a + w/2$. Write $k = \ell(x)$, and let t be the number whose bits after the point are $c(x)$. Truncating m to k digits gives

$$t \le m < t + 2^{-k}. \tag{1}$$

Because $k \ge -\log_2 w + 1$, we have $2^{-k} \le w/2$. So $t > m - 2^{-k} \ge m - w/2 = a$, and $t + 2^{-k} \le m + w/2 = a + w$. This shows that the slice $[t, t + 2^{-k})$, which holds the numbers whose bits begin with $c(x)$, lies inside the final slice of x.
Suppose that $c(x)$ were the beginning of $c(x')$ for two different messages x and x'. Every number whose digits begin with $c(x')$ also begins with $c(x)$, so the slice of $c(x')$ lies inside the slice of $c(x)$. The first lies inside the final slice of x', and the second inside the final slice of x, so these two final slices would share a point. Lemma 0.1 rules that out, so no codeword is the beginning of another. □

Theorem 0.2 (The expected length of an arithmetic code)**.** *Let X be a discrete random variable that takes a finite number of values, each with probability, and let $\mathbb{H}(X)$ be its entropy with base-2 logarithms. Then the arithmetic code for n independent copies of X has expected length*

$$L < n\mathbb{H}(X) + 2, \tag{2}$$

which is fewer than $\mathbb{H}(X) + 2/n$ bits per copy.

Proof

Proof. By Lemma 0.1, the final slice of a message $x = (x_1, \dots, x_n)$ has width $w = \mathbb{p}(x_1) \cdots \mathbb{p}(x_n)$. So the codeword length satisfies

$$\ell(x) = \lceil -\log_2 w \rceil + 1 < -\log_2 w + 2 = \sum_{i=1}^{n} \bigl(-\log_2 \mathbb{p}(x_i)\bigr) + 2. \tag{3}$$

Taking the average over messages, and using that each copy X_i has the expectation $\mathbb{E}\bigl[-\log_2 \mathbb{p}(X_i)\bigr] = \mathbb{H}(X)$, gives $L < n\mathbb{H}(X) + 2$. Dividing by n gives the length per copy. □

Remark 0.1 (Where arithmetic codes come from). Cover and Thomas (2006, sec. 5.9) give Shannon–Fano–Elias coding, which assigns each value a codeword from the midpoint of its slice of $[0, 1)$, in the way that Definition 0.1 does for a whole message. Cover and Thomas describe arithmetic coding, which finds the slice of a long message one value at a time, in section 13.3.

Lempel–Ziv coding

Definition 0.2 (Incremental parsing)**.** Let $x = (x_1, \dots, x_n)$ be a message of symbols. The **incremental parsing** of x cuts it into consecutive *phrases*, reading from the left: each phrase is the shortest string that is not an earlier phrase.
Every phrase is therefore an earlier phrase, or the empty string, followed by one new symbol. The last phrase may be cut short by the end of the message, in which case it repeats an earlier phrase.

Example 0.2 (The incremental parsing of a short message)**.**
The code below follows Definition 0.2 for the message aababababbbbab. It prints the phrases in order, then the number of symbols left over at the end.

```
message_text <- "aababababbbbab"
symbols <- strsplit(message_text, "")[[1]]
phrases <- character(0)
current <- ""
for (symbol in symbols) {
  candidate <- paste0(current, symbol)
  if (candidate %in% phrases) {
    current <- candidate
  } else {
    phrases <- c(phrases, candidate)
    current <- ""
  }
}
phrases
#> [1] "a"   "ab"  "aba" "b"   "ba"  "bb"  "bab"
nchar(current)
#> [1] 0
```

The message has 14 symbols, and the parsing cuts it into 7 phrases with none left over.

Definition 0.3 (Lempel–Ziv code)**.** Let x be a message, and let $c(n)$ be the number of phrases in its incremental parsing. The **Lempel–Ziv code** of x writes each phrase as a pair: the position of the earlier phrase that it extends, with position 0 for the empty string, and the one new symbol.
The code writes each position in $\lceil \log_2 c(n) \rceil$ bits of its own. The code needs no table of phrases to decode: a decoder that reads the pairs in order rebuilds each phrase from the earlier ones. The decoder does need $c(n)$ to know the width of each position, so the encoder sends $c(n)$ first.

Example 0.3 (The Lempel–Ziv code of a short message)**.**
The code below follows Definition 0.3 for the phrases of Example 0.2. It prints each phrase with its pair, then the number of phrases and the length in bits of the code.

```
phrase_length <- nchar(phrases)
prefixes <- substr(phrases, 1, phrase_length - 1)
positions <- match(prefixes, c("", phrases)) - 1
new_symbols <- substr(phrases, phrase_length, phrase_length)
data.frame(phrase = phrases, position = positions, symbol = new_symbols)
#>   phrase position symbol
#> 1      a        0      a
#> 2     ab        1      b
#> 3    aba        2      a
#> 4      b        0      b
#> 5     ba        4      a
#> 6     bb        4      b
#> 7    bab        5      b
n_phrases <- length(phrases)
position_bits <- ceiling(log2(n_phrases))
code_bits <- n_phrases * (position_bits + 1)
c(phrases = n_phrases, code_bits = code_bits)
#>   phrases code_bits
#>         7        28
```

Each position takes 3 bits, and the code adds one more bit for each new symbol, so the code has 28 bits. That is more than the 14 bits of the message itself, so a message this short is too short for the code to save anything.

Theorem 0.3 (The Lempel–Ziv code reaches the entropy)**.** *Let $X_1, X_2, \ldots$ be independent copies of a discrete random variable X that takes a finite number of values, and let $\mathbb{H}(X)$ be its entropy with base-2 logarithms. Let $\ell(x)$ be the number of bits that the Lempel–Ziv code uses for the message $x = (X_1, \ldots, X_n)$. Then with probability one,*

$$\lim_{n\to\infty} \frac{\ell(x)}{n} = \mathbb{H}(X). \tag{4}$$

Proof

Proof. Cover and Thomas (2006, sec. 13.5) prove this for more general sources than this one. The proof is not reproduced here. □

Remark 0.2 (Where the Lempel–Ziv code comes from). Cover and Thomas (2006, sec. 13.4) describe the incremental parsing of this code, and a second form of the code that points back into the message instead. The code needs no knowledge of the probabilities,

> yet Theorem 0.3 shows that it still reaches the entropy.

References

Cover, Thomas M., and Joy A. Thomas. 2006. *Elements of Information Theory*. 2nd ed. Wiley-Interscience. https://doi.org/10.1002/047174882X.