4.4. Arithmetic coding

Advantages:
- Reaches the entropy (within computing precision)
- Superior to Huffman coding for small alphabets and skewed distributions
- Clean separation of modelling and coding
- Suits well for adaptive one-pass compression
- Computationally efficient

History:
- Original ideas by Shannon and Elias
- Actually discovered in 1976 (Pasco; Rissanen)
Arithmetic coding (cont.)

Characterization:
- One codeword for the whole message
- A kind of extreme case of extended Huffman (or Tunstall) coding
- No codebook required
- No clear correspondence between source symbols and code bits

Basic ideas:
- Message is represented by a (small) interval in [0, 1)
- Each successive symbol reduces the interval size
- Interval size = product of symbol probabilities
- Prefix-free messages result in disjoint intervals
- Final code = any value from the interval
- Decoding computes the same sequence of intervals
Arithmetic coding: Encoding of "BADCAB"

A
B
C
D

0.4
0.7
0.9
1

0.4
0.52
0.508
0.52
0.52
etc.
Encoding of "BADCAB" with rescaled intervals
Algorithm: Arithmetic encoding

Input: Sequence $x = x_i$, $i=1, ..., n$; probabilities $p_1, ..., p_q$ of symbols 1, ..., $q$.
Output: Real value between $[0, 1)$ that represents $X$.

begin
  $cum[0] := 0$
  for $i := 1$ to $q$ do $cum[i] := cum[i-1] + p_i$
  lower := 0.0
  upper := 1.0
  for $i := 1$ to $n$ do
    begin
      range := upper − lower
      upper := lower + range * $cum[x_i]$
      lower := lower + range * $cum[x_i-1]$
    end
  return ($lower + upper$) / 2
end
Algorithm: Arithmetic decoding

Input: \( v \): Encoded real value; \( n \): number of symbols to be decoded; probabilities \( p_1, \ldots, p_q \) of symbols \( 1, \ldots, q \).

Output: Decoded sequence \( x \).

begin

\[ \text{cum}[1] := p_1 \]
\[ \text{for } i := 2 \text{ to } q \text{ do } \text{cum}[i] := \text{cum}[i-1] + p_i \]
\[ \text{lower} := 0.0 \]
\[ \text{upper} := 1.0 \]
\[ \text{for } i := 1 \text{ to } n \text{ do } \]

begin range := upper – lower
\[ z := (v – \text{lower}) / \text{range} \]
Find \( j \) such that \( \text{cum}[j-1] \leq z < \text{cum}[j] \)
\[ x_i := j \]
\[ \text{upper} := \text{lower} + \text{range} \times \text{cum}[j] \]
\[ \text{lower} := \text{lower} + \text{range} \times \text{cum}[j-1] \]
end

return \( x = x_1, \ldots, x_n \)
end
Arithmetic coding (cont.)

Practical problems to be solved:
- Arbitrary-precision real arithmetic
- The whole message must be processed before the first bit is transferred and decoded.
- The decoder needs the length of the message

Representation of the final binary code:
- Midpoint between lower and upper ends of the final interval.
- Sufficient number of significant bits, to make a distinction from both lower and upper.
- The code is prefix-free among prefix-free messages.
Example of code length selection

- **upper**: 0.517072 = .1000010001011101...
- **midpoint**: 0.516928 = .10000100010101010...
- **lower**: 0.516784 = .10000100010010111...

```
midpoint ≠ lower and upper
```

```
range = 0.00028
log_2(1/range) ≈ 11.76 bits
```
Another source message

“ABCDABCABA”

- *Precise* probabilities:
  \[ P(A) = 0.4, \ P(B) = 0.3, \ P(C) = 0.2, \ P(D) = 0.1 \]

- Final range length:
  \[ 0.4 \cdot 0.3 \cdot 0.2 \cdot 0.1 \cdot 0.4 \cdot 0.3 \cdot 0.2 \cdot 0.4 \cdot 0.3 \cdot 0.4 = 0.4^4 \cdot 0.3^3 \cdot 0.2^2 \cdot 0.1 = 0.0000002764 \]

  \[-\log_2 0.0000002764 \approx 18.46 = \text{entropy} \]
Arithmetic coding: Basic theorem

Theorem 4.2.
Let $range = upper - lower$ be the final probability interval in Algorithm 4.8. The binary representation of $mid = (upper + lower) / 2$ truncated to $l(x) = \lceil \log_2(1/range) \rceil + 1$ bits is a uniquely decodable code for message $x$ among prefix-free messages.

Proof: Skipped.
Optimality

Expected length of an $n$-symbol message $x$:

$$L^{(n)} = \sum P(x)l(x)$$

$$= \sum P(x) \left[ \log_2 \frac{1}{P(x)} + 1 \right]$$

$$\leq \sum P(x) \left[ \log_2 \frac{1}{P(x)} + 2 \right]$$

$$= \sum P(x) \log_2 \frac{1}{P(x)} + 2 \sum P(x)$$

$$= H(S^{(n)}) + 2$$

Bits per symbol:

$$\frac{H(x^{(n)})}{n} \leq L \leq \frac{H(x^{(n)})}{n} + \frac{2}{n}$$

$$H(S) \leq L \leq H(S) + \frac{2}{n}$$
Ending problem

- The above theorem holds only for prefix-free messages.
- The ranges of a message and its prefix overlap, and may result in the same code value.
- How to distinguish between “VIRTA” and “VIRTANEN”?
- Solutions:
  - Transmit the *length* of the message before the message itself: “5VIRTA” and “8VIRTANEN”.
    This is not good for online applications.
  - Use a special end-of-message symbol, with prob = \(1/n\) where \(n\) is an *estimated* length of the message.
    Good solution unless \(n\) is totally wrong.
Arithmetic coding: Incremental transmission

- Bits are sent as soon as they are known.
- Decoder can start well before the encoder has finished.
- The interval is scaled (zoomed) for each output bit: Multiplication by 2 means shifting the binary point one position to the right:

  - **upper**: $0.011010\ldots$  $\rightarrow$ $0.11010\ldots$ and transmit 0
  - **lower**: $0.001101\ldots$  $\rightarrow$ $0.01101\ldots$

  - **upper**: $0.110100\ldots$  $\rightarrow$ $0.10100\ldots$ and transmit 1
  - **lower**: $0.100011\ldots$  $\rightarrow$ $0.00011\ldots$

- **Note**: The common bit also in midpoint value.
Arithmetic coding: Scaling situations

// Number p of pending bits initialized to 0

**upper < 0.5:**
- transmit bit 0 (plus p pending 1’s)
- \( \text{lower} := 2 \cdot \text{lower} \)
- \( \text{upper} := 2 \cdot \text{upper} \)

**lower > 0.5**
- transmit bit 1 (plus p pending 0’s)
- \( \text{lower} := 2 \cdot (\text{lower} - 0.5) \)
- \( \text{upper} := 2 \cdot (\text{upper} - 0.5) \)

**lower > 0.25 and upper < 0.75:**
- Add one to the number p of pending bits
- \( \text{lower} = 2 \cdot (\text{lower} - 0.25) \)
- \( \text{upper} = 2 \cdot (\text{upper} - 0.25) \)
Decoder operation

- Reads a sufficient number of bits to determine the first symbol (unique interval of cumulative probabilities).
- Imitates the encoder: performs the same scalings, after the symbol is determined
- Scalings drop the ‘used’ bits, and new ones are read in.
- No pending bits.
Implementation with integer arithmetic

- Use symbol frequencies instead of probabilities
- Replace $[0, 1)$ by $[0, 2^{k-1})$
- Replace 0.5 by $2^{k-1}-1$
- Replace 0.25 by $2^{k-2}-1$
- Replace 0.75 by $3\cdot 2^{k-2}-1$

Formulas for computing the next interval:

- $\text{upper} := \text{lower} + (\text{range} \cdot \text{cum}[\text{symbol}] / \text{total}_\text{freq}) - 1$
- $\text{lower} := \text{lower} + (\text{range} \cdot \text{cum}[\text{symbol}-1] / \text{total}_\text{freq})$

Avoidance of overflow: $\text{range} \cdot \text{cum}() < 2^{\text{wordsize}}$

Avoidance of underflow: $\text{range} > \text{total}_\text{frequency}$
Solution to avoiding over-/underflow

- Due to scaling, range is always > $2^{k-2}$
- Both overflow and underflow are avoided, if $total_{freq} < 2^{k-2}$, and $2k-2 \leq w = \text{machine word}$

Suggestion:
- Present $total_{freq}$ with max 14 bits, range with 16 bits

Formula for decoding a symbol $x$ from a $k$-bit value:

$$cum(x - 1) \leq \left\lfloor \frac{(value - lower + 1) \cdot total_{freq} - 1}{upper - lower + 1} \right\rfloor < cum(x)$$
4.4.1. Adaptive arithmetic coding

Advantage of arithmetic coding:
- Used probability distribution can be changed at any time, but synchronously in the encoder and decoder.

Adaptation:
- Maintain frequencies of symbols during the coding
- Use the current frequencies in reducing the interval

Initial model; alternative choices:
- All symbols have an initial frequency = 1.
- Use a placeholder (NYT = Not Yet Transmitted) for the unseen symbols, move symbols to active alphabet at the first occurrence.
Basic idea of adaptive arithmetic coding

- Alphabet: \{A, B, C, D\}
- Message to be coded: “AABAAB …”

<table>
<thead>
<tr>
<th>Alphabet</th>
<th>Frequencies</th>
<th>Interval size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{1,1,1,1}</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>{2,1,1,1}</td>
<td>1/4</td>
</tr>
<tr>
<td>C</td>
<td>{3,1,1,1}</td>
<td>1/10</td>
</tr>
<tr>
<td>D</td>
<td>{3,2,1,1}</td>
<td>1/60</td>
</tr>
<tr>
<td></td>
<td>{4,2,1,1}</td>
<td>3/420</td>
</tr>
</tbody>
</table>

Intervals and Frequencies
Adaptive arithmetic coding (cont.)

Biggest problem:
- Maintenance of cumulative frequencies; simple vector implementation has complexity $O(q)$ for $q$ symbols.

General solution:
- Maintain partial sums in an explicit or implicit binary tree structure.
- Complexity is $O(\log_2 q)$ for both search and update
Tree of partial sums
Implicit tree of partial sums

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$f$</td>
<td>$f_1+f_2$</td>
<td>$f_3$</td>
<td>$f_1+...+f_4$</td>
<td>$f_5$</td>
<td>$f_5+f_6$</td>
<td>$f_7$</td>
<td>$f_1+...+f_8$</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>$f_9$</td>
<td>$f_9+f_{10}$</td>
<td>$f_{11}$</td>
<td>$f_9+...+f_{12}$</td>
<td>$f_{13}$</td>
<td>$f_{13}+f_{14}$</td>
<td>$f_{15}$</td>
<td>$f_1+...+f_{16}$</td>
</tr>
</tbody>
</table>

Correct indices are obtained by bit-level operations.
4.4.2. Arithmetic coding for a binary alphabet

Observations:
- Arithmetic coding works as well for any size of alphabet, contrary to Huffman coding.
- Binary alphabet is especially easy: *No cumulative probability table*.

Applications:
- Compression of black-and-white images
- Any source, interpreted bitwise

Speed enhancement:
- Avoid multiplications
- Approximations cause additional redundancy
Arithmetic coding for binary alphabet (cont.)

Note:
- Scaling operations need only multiplication by two, implemented as shift-left.
- Multiplications appearing in reducing the intervals are the problem.

Convention:
- **MPS** = More Probable Symbol
- **LPS** = Less Probable Symbol
- The correspondence to actual symbols may change locally during the coding.
Skew coder (Langdon & Rissanen)

- **Idea:** approximate the probability $p$ of LPS by $1/2^Q$ for some integer $Q > 0$.

- Choose LPS to be the first symbol of the alphabet (can be done without restriction)

- Calculating the new *range*:
  - For LPS: $\text{range} \leftarrow \text{range} \gg Q$;
  - For MPS: $\text{range} \leftarrow \text{range} - (\text{range} \gg Q)$;

- Approximation causes some redundancy

- Average number of bits per symbol ($p = \text{exact prob}$):
  $$pQ - (1 - p) \log_2 \left(1 - \frac{1}{2^Q}\right)$$
Solving the ‘breakpoint’ probability $\hat{p}$

- Choose $Q$ to be either $r$ or $r+1$, where $r = \lfloor -\log_2 p \rfloor$
- Equate the bit counts for rounding down and up:

$$\hat{p}r - (1 - \hat{p}) \log_2 (1 - \frac{1}{2^r}) = \hat{p}(r + 1) - (1 - \hat{p}) \log_2 (1 - \frac{1}{2^{r+1}})$$

which gives

$$\hat{p} = \frac{z}{1 + z} \quad \text{where} \quad z = \log_2 \frac{1 - 1 / 2^{r+1}}{1 - 1 / 2^r}$$
Skew coder (cont.)

Probability approximation table:

<table>
<thead>
<tr>
<th>Probability range</th>
<th>Q</th>
<th>Effective probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3690 – 0.5000</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>0.1820 – 0.3690</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>0.0905 – 0.1820</td>
<td>3</td>
<td>0.125</td>
</tr>
<tr>
<td>0.0452 – 0.0905</td>
<td>4</td>
<td>0.0625</td>
</tr>
<tr>
<td>0.0226 – 0.0452</td>
<td>5</td>
<td>0.03125</td>
</tr>
<tr>
<td>0.0113 – 0.0226</td>
<td>6</td>
<td>0.015625</td>
</tr>
</tbody>
</table>

Proportional compression efficiency:

\[
\frac{\text{entropy}}{\text{averageLength}} = - p \log p - (1 - p) \log(1 - p) \\

\text{averageLength} = - pQ - (1 - p) \log(1 - 1/2^Q)
\]
QM-coder

- One of the methods for e.g. black-and-white images
- Others:
  - Q-coder (predecessor of QM, tailored to hardware impl. / IBM)
  - MQ-coder (in JBIG2; Joint Bi-Level Image Compression Group)
  - M-coder (in H.264/AVC video compression standard)
- Tuned Markov model (finite-state automaton) for adapting probabilities.

**Interval setting:**
- MPS is the ‘first’ symbol
- Maintain lower and range:

\[
\begin{align*}
\text{lower} + \text{range} \\
\text{lower} + \text{range} \cdot (1-p) \\
\text{range} \cdot p \\
\text{range} \cdot (1-p) \\
\text{lower}
\end{align*}
\]
QM-coder (cont.)

Key ideas:
- Operate within interval \([0, 1.5)\)
- Rescale when \(range < 0.75\)
- Approximate \(range\) by 1 in multiplications
  \[
  range \cdot p \approx p \\
  range \cdot (1-p) \approx range - p
  \]
- No pending bits, but a ‘carry’ bit can propagate to the output bits, which must be buffered. Unlimited propagation is prevented by ‘stuffing’ 0-bits after bytes containing only 1’s (small redundancy).
- Practical implementation is done using integers within \([0, 65536)\).
4.4.3. Practical problems with arithmetic coding

- Not partially decodable nor indexable:
  Start decoding always from the beginning even to recover a small section in the middle.

- Vulnerable: Bit errors result in a totally scrambled message

- Not self-synchronizable, contrary to Huffman code

Solution for static distributions: Arithmetic Block Coding

- Applies the idea of arithmetic coding within machine words
- Restarts a new coding loop when the word bits are ‘used’.
- Resembles Tunstall code, but no explicit codebook.
- Fast, because avoids the scalings and bit-level operations.
- Non-optimal code length, but rather close