Information Theory
Lecture 3

- Lossless source coding algorithms:
  - Huffman: CT5.6–8
  - Shannon-Fano-Elias: CT5.9
  - Arithmetic: CT13.3
  - Lempel-Ziv: CT13.4–5

Example: Encoding a Markov Source

- 2-state Markov chain \( P_{01} = P_{10} = \frac{1}{3} \implies \mu_0 = \mu_1 = \frac{1}{2} \)
- Sample sequence
  \[ s = 1000011010001111 = 1\ 0\ 4\ 1\ 2\ 0\ 1\ 3\ 1\ 4 \]
- Probabilities of 2-bit symbols
  \[
  \begin{array}{c|c|c|c|c|c}
  & p(00) & p(01) & p(10) & p(11) & H  \\ \hline
  \text{sample} & \frac{1}{4} & \frac{1}{6} & \frac{2}{6} & \frac{1}{3} & 1.9056 \\ \hline
  \text{model} & \frac{1}{3} & \frac{1}{6} & \frac{2}{6} & \frac{1}{3} & 1.9183 \\
  \end{array}
  \]
- Entropy rate
  \[ H(S) = h(\frac{1}{3}) \approx 0.9183 \implies L \geq \lceil 14.6928 \rceil = 15 \]

Zero-Error Source Coding

- Huffman codes: algorithm & optimality
- Shannon-Fano-Elias codes
  - connection to Shannon(-Fano) codes, Fano codes, and per symbol arithmetic coding
  - within 2(1) symbol of the entropy
- Arithmetic codes
  - adaptable, probabilistic model
  - within 2 bits of the entropy per sequence!
- Lempel-Ziv codes
  - “basic” and “modified” LZ-algorithm
  - sketch of asymptotic optimality

Huffman Coding Algorithm

- Greedy bottom-up procedure
- Builds a complete \( D \)-ary codetree by combining the \( D \) symbols of lowest probabilities
  \( \Rightarrow \) need \( |\mathcal{X}| = 1 \mod D - 1 \)
  \( \Rightarrow \) add dummy symbols of 0 probability if necessary
- Gives prefix code
- Probabilities of source symbols need to be available
  \( \Rightarrow \) coding long strings (“super symbols”) becomes complex
Huffman Code Examples

<table>
<thead>
<tr>
<th>sample-based</th>
<th>model-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>11: (\frac{1}{4})</td>
<td>11: (\frac{1}{5})</td>
</tr>
<tr>
<td>10: (\frac{3}{8})</td>
<td>00: (\frac{1}{3})</td>
</tr>
<tr>
<td>01: (\frac{1}{8})</td>
<td>10: (\frac{1}{8})</td>
</tr>
<tr>
<td>00: (\frac{1}{4})</td>
<td>01: (\frac{1}{8})</td>
</tr>
</tbody>
</table>

16, \(|1000001110000101| = 16\) 16, \(|0010100001010111| = 18\)

Optimal Symbol Codes

- An optimal binary prefix code must satisfy

  \[ p(x) \leq p(y) \implies l(x) \geq l(y) \]

- There are at least two codewords of maximal length
- The longest codewords can be relabeled such that the two least probable symbols differ only in their last bit
- **Huffman codes are optimal prefix codes** (why?)
  - We know that
    \[ L = H(X) \iff l(x) = -\log p(x) \]
    \[ \implies \text{Huffman will give } L = H(X) \text{ when } -\log p(x) \text{ are integers} \]
    (a dyadic distribution)

Cumulative Distributions and Rounding

- \( X \in \mathcal{X} = \{1, 2, \ldots, m\}; \ p(x) = \Pr(X = x) > 0 \)
- Cumulative distribution function (cdf)
  \[ F(x) = \sum_{x' \leq x} p(x'), \quad x \in [0, m] \]
- Modified cdf
  \[ \bar{F}(x) = \sum_{x' < x} p(x') + \frac{1}{2} p(x), \quad x \in \mathcal{X} \]
  - only for \( x \in \mathcal{X} \)
  - \( \bar{F}(x) \) known \( \implies x \) known!

- We know that \( l(x) \approx -\log p(x) \) gives a good code
- Use the binary expansion of \( \bar{F}(x) \) as code for \( x \); rounding needed
  - round to \( \approx -\log p(x) \) bits
- Rounding: \([0, 1) \rightarrow \{0, 1\}^k\)
  - Use base 2 fractions
    \[ f \in [0, 1) \implies f = \sum_{i=1}^{\infty} f_i 2^{-i} \]
  - Take the first \( k \) bits
    \[ [f]_k = f_1 f_2 \cdots f_k \in \{0, 1\}^k \]
    - For example, \( \frac{3}{8} = 0.10101010 \cdots = 0.\overline{10} \implies [\frac{3}{8}]_5 = 10101 \)
**Shannon-Fano-Elias Codes**

- Shannon-Fano-Elias code (as it is described in CT)
  
  \[ l(x) = \lceil \log \frac{1}{p(x)} \rceil + 1 \quad \Rightarrow \quad L < H(X) + 2 \text{ [bits]} \]

- \( c(x) = \lfloor \bar{F}(x) \rfloor l(x) = \lfloor F(x) + \frac{1}{2}p(x) \rfloor l(x) \)

- *Prefix-free* if intervals \([0, c(x)], 0, c(x) + 2^{-l(x)}\) disjoint (why?)

\[ \Rightarrow \quad \text{instantaneous code (check)} \]

- Example:

<table>
<thead>
<tr>
<th>( X )</th>
<th>( p(x) )</th>
<th>( l(x) )</th>
<th>( \bar{F}(x) )</th>
<th>( c(x) )</th>
<th>( p(x) )</th>
<th>( l(x) )</th>
<th>( \bar{F}(x) )</th>
<th>( c(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1(00)</td>
<td>1/4</td>
<td>3</td>
<td>1/8</td>
<td>001</td>
<td>1/3</td>
<td>3</td>
<td>1/6</td>
<td>001</td>
</tr>
<tr>
<td>2(01)</td>
<td>1/8</td>
<td>4</td>
<td>5/16</td>
<td>0101</td>
<td>1/6</td>
<td>4</td>
<td>5/12</td>
<td>0110</td>
</tr>
<tr>
<td>3(10)</td>
<td>3/8</td>
<td>3</td>
<td>9/16</td>
<td>100</td>
<td>1/6</td>
<td>4</td>
<td>7/12</td>
<td>1001</td>
</tr>
<tr>
<td>4(11)</td>
<td>1/4</td>
<td>3</td>
<td>7/8</td>
<td>111</td>
<td>1/3</td>
<td>3</td>
<td>5/6</td>
<td>110</td>
</tr>
</tbody>
</table>

\[ L = 3.125 < H(X) + 2 \quad L = 3.333 < H(X) + 2 \]

**Intervals**

- Dyadic intervals

  - A binary string can represent a subinterval of \([0, 1)\)

  \[ x_1x_2 \cdots x_m \in \{0, 1\}^m \quad \Rightarrow \quad x = \sum_{i=1}^{m} x_i 2^{-m-i} \in \{0, 1, \ldots, 2^m - 1\} \]

  (the usual binary representation of \( x \)), then

  \[ x_1x_2 \cdots x_m \rightarrow \left[ \frac{x}{2^m}, \frac{x + 1}{2^m} \right) \subset [0, 1) \]

  - For example, \( 110 \rightarrow \left[ \frac{3}{8}, \frac{7}{8} \right) \)

**Arithmetic Coding – Symbol**

- “Algorithm”
  
  - No preset codeword lengths for rounding off
  
  - Instead, *the largest dyadic interval inside the symbol interval* gives the codeword for the symbol

- Example: Shannon-Fano-Elias vs. arithmetic symbol code

<table>
<thead>
<tr>
<th>( X )</th>
<th>sample-based</th>
<th>model-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>11</td>
<td>1101</td>
<td>0101</td>
</tr>
<tr>
<td>00</td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>
Arithmetic Coding – Stream

- Works for streams as well (main advantage over e.g. Huffman)!
- But, how do we know when to stop decoding?
  - Use a large, but fixed, source block length \( N \) (CT)
  - Introduce a special “termination” symbol
    (slightly skews statistics, but handles any source length)
- Consider binary strings, for simplicity, and order strings according to their corresponding integers (e.g., \( 0111 < 1000 \)), let
  \[
  F(x_1^N) = \sum_{y_1^N \leq x_1^N} \Pr(X_1^N = y_1^N) = \sum_{k: x_k = 1} p(x_1 x_2 \cdots x_{k-1} 0) + p(x_1^N)
  \]

  *Sum over all strings to the left of \( x_1^N \) in a binary tree
  (with 00 \cdots 0 to the far left)*

Arithmetic Coding – Adaptive

- Only the distribution of the current symbol conditioned on the past symbols is needed at every step
  \( \Rightarrow \) Easily made adaptive: just estimate \( p(x_{n+1} | x_1^n) \)
- One such estimate is given by the Laplace model
  \[
  \Pr(x_{n+1} = x | x_1^n) = \frac{n_x + 1}{n + |x'|}
  \]

Lempel-Ziv: A Universal Code

- Not a symbol code
- Quite another philosophy: parsings, phrases, dictionary
- A parsing divides \( x_1^n \) into phrases \( y_1^{c(n)} \)
  \[
  x_1 x_2 \cdots x_n \rightarrow y_1, y_2, \ldots, y_{c(n)}
  \]
- In a distinct parsing phrases do not repeat
  - The LZ algorithm performs a greedy distinct parsing, whereby each new phrase extends an old phrase by just 1 bit
  \( \Rightarrow \) The LZ code for the new phrase is simply the dictionary index of the old phrase followed by the extra bit
- There are several variants of LZ coding, we consider the “basic” and the “modified” LZ algorithms
The “Basic” Lempel-Ziv Algorithm

- Lempel-Ziv parsing and “basic” encoding of $s$

<table>
<thead>
<tr>
<th>phrases</th>
<th>$\lambda$</th>
<th>1</th>
<th>0</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>100</th>
<th>011</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>indices</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>encoding</td>
<td>01</td>
<td>0,</td>
<td>0</td>
<td>10,</td>
<td>10,</td>
<td>001,</td>
<td>010,</td>
<td>100,</td>
<td>001,</td>
</tr>
</tbody>
</table>

- Remarks
  - Parsing starts with empty string
  - First pointer sent is also empty
  - Only “important” index bits are used
  - Even so, “compressed” 16 bits to 24 bits

The “Modified” Lempel-Ziv Algorithm

- The second time a phrase occurs,
  - the extra bit is known
  - it cannot be extended a distinct third way
  $\Rightarrow$ the second extension may overwrite the parent

- Lempel-Ziv parsing and “modified” encoding of $s$

<table>
<thead>
<tr>
<th>phrases</th>
<th>$\lambda$</th>
<th>1</th>
<th>0</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>100</th>
<th>011</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>indices</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>encoding</td>
<td>01</td>
<td>0,</td>
<td>0</td>
<td>0,</td>
<td>01,</td>
<td>11,</td>
<td>000,</td>
<td>001,</td>
<td>001,</td>
</tr>
</tbody>
</table>

$\Rightarrow$ saved 5 bits! (still 16:19 “compression”)

Asymptotic Optimality of LZ Coding

- Codeword lengths of Lempel-Ziv codes satisfy (index + extra bit)

$$l(x^n_i) \leq c(n)(\log c(n) + 1)$$

- Using a counting argument, the number of phrases $c(n)$ in a distinct parsing of a length $n$ sequence is bounded as

$$c(n) \leq \frac{n}{\log n} (1 + o(1))$$

- Ziv’s lemma relates distinct parsings and a $k^{th}$-order Markov approximation of the underlying distribution.

- Combining the above leads to the optimality result:
  - For a stationary and ergodic source $\{X_n\}$,

$$\limsup_{n \to \infty} \frac{1}{n} l(X^n_i) \leq H(S) \quad \text{a.s.}$$
Generating Discrete Distributions from Fair Coins

- A natural inverse to data compression
- Source encoders aim to produce i.i.d. fair bits (symbols)
- Source decoders noiselessly reproduce the original source sequence (with the proper distribution)

⇒ "Optimal" source decoders provide an efficient way to generate discrete random variables