Outline

• Definition of Entropy
• Three Entropy coding techniques:
  • Huffman coding
  • Arithmetic coding
  • Lempel-Ziv coding

Entropy Coding
(taken from the Technion)

Entropy

Entropy of a set of elements $e_1, \ldots, e_n$ with probabilities $p_1, \ldots, p_n$ is:

$$H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

**Entropy** (in our context) - smallest number of bits needed, on the average, to represent a symbol (the average on all the symbols code lengths).

Note: $\log_2 p_i$ is the uncertainty in symbol $e_i$ (or the "surprise" when we see this symbol). Entropy – average "surprise"

Assumption: there are no dependencies between the symbols’ appearances

Definitions

**Alphabet**: A finite set containing at least one element: $A = \{a, b, c, d, e\}$

**Symbol**: An element in the alphabet: $s \in A$

**A string over the alphabet**: A sequence of symbols, each of which is an element of that alphabet: ccdabcdaad…

**Codeword**: A sequence of bits representing a coded symbol or string: 110101001101010100…

$p_i$: The occurrence probability of symbol $s_i$ in the input string. $\sum_{i=1}^{\infty} p_i = 1$

$L_i$: The length of the codeword of symbol $s_i$ in bits.
Entropy examples

• Entropy of $e_1, \ldots, e_n$ is maximized when
  \[ p_1 = p_2 = \ldots = p_n = \frac{1}{n} \Rightarrow H(e_1, \ldots, e_n) = \log_2 n \]
  - No symbol is "better" than the other or contains more information
  - $2^n$ symbols may be represented by $k$ bits
• Entropy of $p_1, \ldots, p_n$ is minimized when
  \[ p_1 = 1, p_2 = \ldots = p_n = 0 \Rightarrow H(e_1, \ldots, e_n) = 0 \]

Entropy calculation for a two symbol alphabet.

Example 1:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.5</td>
</tr>
<tr>
<td>B</td>
<td>0.5</td>
</tr>
</tbody>
</table>

\[ H(A, B) = -p_A \log_2 p_A - p_B \log_2 p_B \]
\[ = -0.5 \log_2 0.5 - 0.5 \log_2 0.5 = 1 \]

It requires one bit per symbol on the average to represent the data.

Example 2:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.8</td>
</tr>
<tr>
<td>B</td>
<td>0.2</td>
</tr>
</tbody>
</table>

\[ H(A, B) = -p_A \log_2 p_A - p_B \log_2 p_B \]
\[ = -0.8 \log_2 0.8 - 0.2 \log_2 0.2 \approx 0.7219 \]

It requires less than one bit per symbol on the average to represent the data.

How can we code this?

Entropy examples

• Entropy of $e_1, \ldots, e_n$ is maximized when
  \[ p_1 = p_2 = \ldots = p_n = \frac{1}{n} \Rightarrow H(e_1, \ldots, e_n) = \log_2 n \]

• No symbol is "better" than the other or contains more information
  - $2^n$ symbols may be represented by $k$ bits

• Entropy of $p_1, \ldots, p_n$ is minimized when
  \[ p_1 = 1, p_2 = \ldots = p_n = 0 \Rightarrow H(e_1, \ldots, e_n) = 0 \]

Code types

• **Fixed-length codes** - all codewords have the same length (number of bits)
  - A – 000, B – 001, C – 010, D – 011, E – 100, F – 101

• **Variable-length codes** - may give different lengths to codewords
  - A – 0, B – 00, C – 110, D – 111, E – 1000, F – 1011

Entropy coding

• Entropy is a lower bound on the average number of bits needed to represent the symbols (the data compression limit).

• Entropy coding methods:
  - Aspire to achieve the entropy for a given alphabet, BPS > Entropy
  - A code achieving the entropy limit is optimal

\[ \text{BPS} = \frac{\text{encoded message}}{\text{original message}} \]
**Huffman coding**

- Each symbol is assigned a variable-length code, depending on its frequency. The higher its frequency, the shorter the codeword.
- Number of bits for each codeword is an integral number.
- A prefix code.
- A variable-length code.
- Huffman code is the optimal prefix and variable-length code, given the symbols' probabilities of occurrence.
- Codewords are generated by building a Huffman Tree.

**Code types (cont.)**

- **Prefix code** - No codeword is a prefix of any other codeword.
  
  \[A = 0; \ B = 10; \ C = 110; \ D = 111\]

- **Uniquely decodable code** - Has only one possible source string producing it.
  - Unambiguously decoded.
  - Examples:
    - Prefix code - the end of a codeword is immediately recognized without ambiguity: 01001101001110

**Huffman encoding**

Use the codewords from the previous slide to encode the string "BCAE":

- **String:** B C A E
- **Encoded:** 10 00 01 111
- **Number of bits used:** 9

The BPS is (9 bits/4 symbols) = 2.25

**Entropy:**

- \(-0.25\log_{2}0.25 - 0.25\log_{2}0.25 - 0.2\log_{2}0.2 - 0.15\log_{2}0.15 - 0.15\log_{2}0.15\) = 2.2854

BPS is lower than the entropy. **WHY?**

**Huffman tree example**

- Each codeword is determined according to the path from the root to the symbol.
- When decoding, a tree traversal is performed, starting from the root.

Example: decoding input "110" (D)

- Probabilities:
  - 0.26, 0.2, 0.26, 0.15, 0.15

- Codewords:
  - A-01, C-00, B-10, D-110, E-111
**Huffman tree construction**

- **Initialization:**
  - Leaf for each symbol \( x \) of alphabet \( A \) with weight \( p_x \).
  - Note: One can work with integer weights in the leaves (for example, number of symbol occurrences) instead of probabilities.
  - **while** (tree not fully connected) do begin
    - \( Y, Z \leftarrow \text{lowest_root_weights_tree}() \)
    - \( r \leftarrow \text{new_root} \)
    - \( r->\text{attachSons}(Y, Z) \) // attach one via a 0, the other via a 1 (order not significant)
    - \( \text{weight}(r) = \text{weight}(Y)+\text{weight}(Z) \)

**Symbol probabilities**

- How are the probabilities known?
  - Counting symbols in input string
    - Data must be given in advance
    - Requires an extra pass on the input string
  - Data source’s distribution is known
    - Data not necessarily known in advance, but we know its distribution

**Huffman encoding**

- Build a table of per-symbol encodings (generated by the Huffman tree).
  - Globally known to both encoder and decoder
  - Sent by encoder, read by decoder
- Encode one symbol after the other, using the encoding table.
- Encode the pseudo-eof symbol.

**Huffman decoding**

- Construct decoding tree based on encoding table
- Read coded message bit-by-bit:
  - Travers the tree top to bottom accordingly.
  - When a leaf is reached, a codeword was found → corresponding symbol is decoded
  - Repeat until the pseudo-eof symbol is reached.

No ambiguities when decoding codewords (prefix code)
### Huffman Entropy analysis

Best results (entropy wise) - only when symbols have occurrence probabilities which are negative powers of 2 (i.e. \( \frac{1}{2}, \frac{1}{4}, \ldots \)). Otherwise, BPS > entropy bound.

**Example:**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0.25</td>
<td>01</td>
</tr>
<tr>
<td>C</td>
<td>0.125</td>
<td>001</td>
</tr>
<tr>
<td>D</td>
<td>0.125</td>
<td>000</td>
</tr>
</tbody>
</table>

Entropy = **1.75**

A representing probabilities input stream: AAAABBCD

Code: 1110101001000

BPS = (14 bits/8 symbols) = **1.75**

### Huffman tree construction complexity

- Simple implementation - \( o(n^2) \).
- Using a Priority Queue - \( o(n \log(n)) \):
  - Inserting a new node – \( o(\log(n)) \)
  - \( n \) nodes insertions - \( o(n \log(n)) \)
  - Retrieving 2 smallest node weights – \( o(\log(n)) \)
Arithmetic coding

- Assigns one (normally long) codeword to entire input stream
- Reads the input stream symbol by symbol, appending more bits to the codeword each time
- Codeword is a number, representing a segmental subsection based on the symbols’ probabilities
- Encodes symbols using a non-integer number of bits → very good results (entropy wise)

Mathematical definitions

L – The smallest binary value consistent with a code representing the symbols processed so far.

R – The product of the probabilities of those symbols.
Arithmetic encoding (cont.)

Two possibilities for the encoder to signal to the decoder end of the transmission:

1. Send initially the number of symbols encoded.
2. Assign a new EOF symbol in the alphabet, with a very small probability, and encode it at the end of the message.

Note: The order of the symbols in the alphabet must remain consistent throughout the algorithm.

Arithmetic decoding example

Decoding of 0.386

Arithmetic decoding

In order to decode the message, the symbols order and probabilities must be passed to the decoder.

The decoding process is identical to the encoding. Given the codeword (the final number), at each iteration the corresponding sub-range is entered, decoding the symbol representing the specific range.
Arithmetic entropy analysis

- Arithmetic coding manages to encode symbols using non integer number of bits!
- One codeword is assigned to the entire input stream, instead of a codeword to each individual symbol
- This allows Arithmetic Coding to achieve the Entropy lower bound

Distributions issues

- Until now, symbol distributions were known in advance
- What happens if they are not known?
  - Input string not known
  - Huffman and Arithmetic Codings have an adaptive version
    - Distributions are updated as the input string is read
    - Can work online

Lempel-Ziv concepts

- What if the alphabet is unknown? Lempel-Ziv coding solves this general case, where only a stream of bits is given.
- LZ creates its own dictionary (strings of bits), and replaces future occurrences of these strings by a shorter position string:
  - In simple Huffman/Arithmetic coding, the dependency between the symbols is ignored, while in the LZ, these dependencies are identified and are exploited to perform better encoding.
  - When all the data is known (alphabet, probabilities, no dependencies), it’s best to use Huffman (LZ will try to find dependencies which are not there...)

Lempel-Ziv concepts
**Lempel-Ziv compression**

- Parses source input (in binary) into the shortest distinct strings: 1011010100010 → 1, 0, 11, 01, 010, 00, 10
- Each string includes a prefix and an extra bit (010 = 01 + 0), therefore encoded as: (prefix string place, extra bit)
- Requires 2 passes over the input (one to parse input, second to encode). Can be modified to one pass.
- Compression: \( n \cdot \log(n) \) bits compressed

**Example**

Input string: 1 0 1 1 0 1 0 0 1 0

```
<table>
<thead>
<tr>
<th>W</th>
<th>Ø</th>
<th>Ø</th>
<th>1</th>
<th>0</th>
<th>01</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Index  | Entry
0      | Ø
1      | 1
2      | 0
3      | 11
4      | 01
5      | 010
6      | 00
7      | 10
```

Encoded string:

```
0001 0000 0011 0101 0000 0100 0000 0000 0010 0000 0000
```

**Compression comparison**

<table>
<thead>
<tr>
<th>Compressed to (percentage)</th>
<th>Lempel-Ziv (unix gzip)</th>
<th>Huffman (unix pack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>html (25k) Token based ascii file</td>
<td>20%</td>
<td>65%</td>
</tr>
<tr>
<td>pdf (690k) Binary file</td>
<td>75%</td>
<td>95%</td>
</tr>
<tr>
<td>ABCD (1.5k) Random ascii file</td>
<td>33%</td>
<td>28.2%</td>
</tr>
<tr>
<td>ABCD (500k) Random ascii file</td>
<td>29%</td>
<td>28.1%</td>
</tr>
</tbody>
</table>

ABCD – (\( p_A = 0.5, p_B = 0.25, p_C = 0.125, p_D = 0.125 \))

Lempel-Ziv is asymptotically optimal

**Lempel-Ziv algorithm**

1. Initialize the dictionary to contain an empty string (D=Ø).
2. \( W \leftarrow \) longest block in input string which appears in D.
3. \( B \leftarrow \) first symbol in input string after W
4. Encode W by its index in the dictionary, followed by B
5. Add \( W+B \) to the dictionary.
6. Go to Step 2.
### Comparison

<table>
<thead>
<tr>
<th></th>
<th>Huffman</th>
<th>Arithmetic</th>
<th>Lempel-Ziv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probabilities</td>
<td>Known in advance</td>
<td>Known in advance</td>
<td>Not known in advance</td>
</tr>
<tr>
<td>Alphabet</td>
<td>Known in advance</td>
<td>Known in advance</td>
<td>Not known in advance</td>
</tr>
<tr>
<td>Data loss</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Symbols dependency</td>
<td>Not used</td>
<td>Not used</td>
<td>Used – better compression</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>Tree building – O(n log n)</td>
<td>None</td>
<td>First pass on data (can be eliminated)</td>
</tr>
<tr>
<td>Entropy</td>
<td>If probabilities are negative powers of 2</td>
<td>Very close</td>
<td>Best results when alphabet not known</td>
</tr>
<tr>
<td>Codewords</td>
<td>One codeword for each symbol</td>
<td>One codeword for all data</td>
<td>Codewords for set of alphabet</td>
</tr>
<tr>
<td>Intuition</td>
<td>Intuitive</td>
<td>Not intuitive</td>
<td>Not intuitive</td>
</tr>
</tbody>
</table>