Entropy Coding

Connectivity coding

Connectivity data

Add 7, Add 6, Add 7, Add 5,...

TG output

CRRRLSLECRRE

Entropy coder output

Edgebreaker output

Entropy coding

Lossless coder

Input: a set of symbols

Output: bitstream

Idea

Assign each symbol a series of bits

Use less bits for common symbols

Definitions

**Alphabet**
Finite set containing at least one element

**Symbol**
Element in the alphabet

**A string over the alphabet**
Sequence of symbols from alphabet

**Codeword**
Bits representing coded symbol or string

**p_i**
Occurrence probability of s_i in input string

**L_i**
Length of codeword of s_i in bits

\[ A = \{a, b, c, d, e\} \]

\[ s_i \in A \]

\[ S = ccdabcdaad \]

\[ 110101001101010100 \]

\[ p_i = P(s_i \in S), \sum_{i=1}^{n} p_i = 1 \]
Entropy

Entropy of the set \( \{ e_1, \ldots, e_n \} \) with probabilities \( \{ p_1, \ldots, p_n \} \)

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

- \( \log_2 p_i \) = uncertainty in symbol \( e_i \)
  - The “surprise” when we see this symbol
  - Entropy – average “surprise” on all symbols

In our context
- Minimal number of bits on the average, needed to represent a symbol
- Average on all symbols code lengths
- Assuming no dependencies between symbols’ appearances

Entropy example 1

Entropy calculation for a two symbol alphabet.

Example 1:  
\[ A \quad p_A=0.5 \]
\[ B \quad p_B=0.5 \]

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

We need 1 bit per symbol on average to represent the data.

Entropy example 2

Entropy calculation for a two symbol alphabet.

Example 1:  
\[ A \quad p_A=0.8 \]
\[ B \quad p_B=0.2 \]

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

We need LESS than 1 bit per symbol on average.

Entropy examples

- Entropy of \( \{ e_1, \ldots, e_n \} \) is maximized when
  \[ p_1=p_2=\ldots=p_n=1/n \quad \rightarrow \quad H(e_1, \ldots, e_n)=\log_2 n \]
  - No symbol is “better” than the other or contains more information
  - \( 2^k \) symbols must be represented by \( k \) bits

- Entropy of \( \{ e_1, \ldots, e_n \} \) is minimized when
  \[ p_1=1, p_2=\ldots=p_n=0 \quad \rightarrow \quad H(e_1, \ldots, e_n)=0 \]
Entropy coding

- Entropy
  - Lower bound on average number of bits needed for alphabet
  - Data compression limit

- Coding efficiency = Bits Per Symbol
  \[ \text{BPS} = \frac{\text{length(encoded message)}}{\text{length(original message)}} \]

- Entropy coding methods
  - Try to achieve entropy of alphabet: BPS → Entropy
  - If BPS = Entropy, code is optimal

Code types

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

- **Variable-length codes**
  - Codewords can have different lengths
  - A – 0, B – 00, C – 110, D – 111, E – 1000, F – 1011

Huffman code

- A variable-length prefix code
  - Codeword chosen by probability of appearance
    - High probability → short codeword
  - Integral number of bits per codeword
  - **Optimal** variable-length prefix code for known probabilities
  - Encoding/decoding done using Huffman tree
**Huffman tree example**

 Codeword determined according to path from root to symbol

 When decoding, tree traversal is performed, starting from root

**Codewords**

- A-01
- C-00
- B-10
- D-110
- E-111

**Example:**

Decoding input “110” (D)

**Huffman encoding example**

Use previous codewords to encode “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: -0.25log0.25 - 0.25log0.25 - 0.2log0.2 - 0.15log0.15 - 0.15log0.15 = 2.2854

**BPS lower than entropy. WHY?**

**Huffman tree construction**

- **Init:**
  - Leaf for each symbol \( s \) of alphabet \( A \) with weight \( p_s \)

- while (tree not connected) do
  - \( Y, Z \) \( \leftarrow \) lowest_root_weights()
  - \( r \leftarrow \) new_root
  - \( r \rightarrow \) attachSons(\( Y, Z \))
  - \( \text{weight}(r) = \text{weight}(Y) + \text{weight}(Z) \)

**Probabilities**

- 0.25
- 0.2
- 0.25
- 0.15
- 0.15

**Huffman tree construction**

- **Initialization**
  - Leaf for each symbol \( s \) of alphabet \( A \) with weight \( p_s \)
  - Can work instead with integer weights - number of occurrences

- while (tree not connected) do
  - \( Y, Z \) \( \leftarrow \) lowest_root_weights_tree()
  - \( r \leftarrow \) new_root
  - \( r \rightarrow \) attachSons(\( Y, Z \))
  - attach one via a 0, the other via a 1, order not significant
  - \( \text{weight}(r) = \text{weight}(Y) + \text{weight}(Z) \)
Huffman encoding

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

Huffman decoding

- Construct decoding tree based on encoding table
- Read coded message bit-by-bit
  - Traverse 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 - prefix code

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 known in advance, but distribution is known

Example

"Global" English frequencies table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.0721</td>
<td>N</td>
<td>0.0638</td>
</tr>
<tr>
<td>B</td>
<td>0.0240</td>
<td>O</td>
<td>0.0681</td>
</tr>
<tr>
<td>C</td>
<td>0.0390</td>
<td>P</td>
<td>0.0290</td>
</tr>
<tr>
<td>D</td>
<td>0.0372</td>
<td>Q</td>
<td>0.0023</td>
</tr>
<tr>
<td>E</td>
<td>0.1224</td>
<td>R</td>
<td>0.0638</td>
</tr>
<tr>
<td>F</td>
<td>0.0272</td>
<td>S</td>
<td>0.0728</td>
</tr>
<tr>
<td>G</td>
<td>0.0178</td>
<td>T</td>
<td>0.0908</td>
</tr>
<tr>
<td>H</td>
<td>0.0449</td>
<td>U</td>
<td>0.0235</td>
</tr>
<tr>
<td>I</td>
<td>0.0779</td>
<td>V</td>
<td>0.0094</td>
</tr>
<tr>
<td>J</td>
<td>0.0013</td>
<td>W</td>
<td>0.0130</td>
</tr>
<tr>
<td>K</td>
<td>0.0054</td>
<td>X</td>
<td>0.0077</td>
</tr>
<tr>
<td>L</td>
<td>0.0426</td>
<td>Y</td>
<td>0.0126</td>
</tr>
<tr>
<td>M</td>
<td>0.0282</td>
<td>Z</td>
<td>0.0026</td>
</tr>
</tbody>
</table>

Total: 1.0000
Huffman entropy analysis

Best results - entropy wise
- Only when occurrence probabilities 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

An input stream which represents the probabilities
- AAAABBCD Code: 1110101001000

BPS = (14 bits/8 symbols) = 1.75

Huffman tree

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

Huffman summary

- Achieves entropy when occurrence probabilities are negative powers of 2
- Alphabet and distribution must be known in advance
- Given Huffman tree, very easy (and fast) to encode and decode
- Huffman code not unique (arbitrary decisions in tree construction)

Better than Huffman?

- Huffman optimal, so how can improve?
- Use fractional number of bits per codeword
  - Arithmetic coding
- Learn probabilities from bit stream
  - Lempel-Ziv coding
    - Unknown alphabet
    - Unknown probabilities
    - Handles dependencies between symbols