Is Huffman optimal?

We know a prefix code will have $H(X) \leq L(X) < H(X)+1$

Huffman is a prefix code, but is there a better one, nearer $H(X)$??

Lemma 1 More probable symbols have shorter codewords.

Proof by contradiction.

Imagine an optimal tree where $p_i > p_j$ but $l_i > l_j$

Let's swap C_i and C_j codewords

Original $L = \ldots + p_i l_i + p_j l_j + \ldots$

New $L' = \ldots + p_i l_j + p_j l_i + \ldots$

$$L' - L = (p_i - p_j)(l_j - l_i)$$

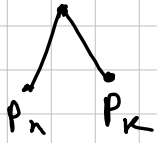
>0 <0

$\therefore$ $L' - L < 0 \Rightarrow$ contradicts L being optimal. $\square$

Lemma 2 The two least probable symbols are siblings at the deepest level.

Proof.

- Let p_{n-1} and p_n belong to the least probable symbols
- Lemma 1 $\Rightarrow$ p_n and p_{n-1} must be at the maximum depth
- Suppose p_n is at max depth with sibling $p_k \neq p_{n-1}$



- $p_k \geq p_{n-1}$ since p_{n-1} and p_n are the smallest.
- $l_k = l_{n-1}$ because p_{n-1} and p_n have same max depth
- We can swap p_k and p_{n-1} at cost $(p_k - p_{n-1})(l_{n-1} - l_k)$
  $= (p_k - p_{n-1})(0) = 0$
- Now p_n and p_{n-1} are siblings.

$\Rightarrow$ We can always construct an optimal tree where the two least probable symbols are siblings

□

Optimality Proof for Huffman Codes

Proof by induction

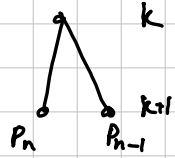
Base case: $n=2$. Huffman assigns '0' and '1' which is trivially optimal.

Inductive Hypothesis: Huffman is optimal for any set of $(n-1)$ symbols.

Inductive step:

- Given n symbols, Huffman takes the two lowest probability, P_{n-1} and P_n, and merges them $p' = P_{n-1} + P_n$
- There are now $(n-1)$ symbols. Huffman constructs an optimal tree, T', by the inductive hypothesis
- In final tree, T, P_{n-1} and P_n are siblings (Lemma 2)

  $\therefore$ $l_n = k+1$, $l_{n-1} = k+1$ for parent at depth k



- In T, $l' = k$ for the merged symbol

- $L(T) = \sum_{i=1}^{n-2} p_i l_i + p_{n-1} l_{n-1} + p_n l_n$

$$= \sum p_i l_i + p_{n-1}(k+1) + p_n(k+1)$$

$$= \sum p_i l_i + (p_{n-1} + p_n)(k+1)$$

$$= \sum p_i l_i + p'(k+1)$$

$$= \sum p_i l_i + kp' + p'$$

$$= \sum p_i l_i + p'l' + p'$$

$$= L(T') + p'$$

$$= L(\bar{T}') + \underbrace{(p_{n-1} + p_n)}_{\text{Constant}}$$

→ Expected length of T is the expected length of T' plus a constant.

→ To minimise $L(\bar{T})$ we minimise $L(T')$

→ Huffman minimises $L(T')$.

- Huffman constructs the optimal tree

□

Recap

Prefix codes are instantly decodable

Huffman coding is an optimal prefix code.
for a given symbol

Prefix codes achieve $H(x) \leq L(x) < H(x)+1$

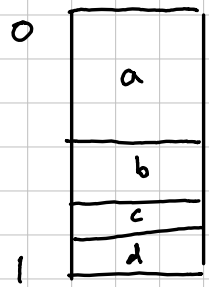
Entropy seems to represent the core information content of a message – we will look at this soon.

Q. Is there a better approach than prefix codes?

[Dasher demo]

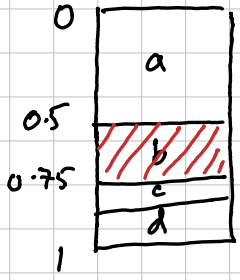
Arithmetic Coding

a	1/2
b	1/4
c	1/8
d	1/8



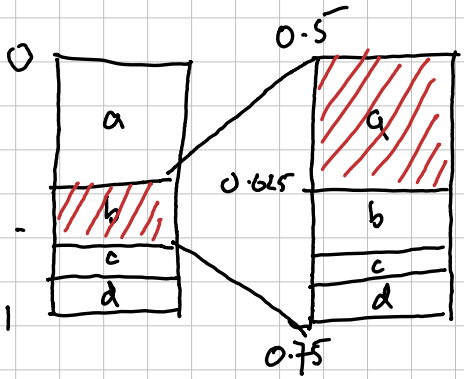
A.C. represents a set of symbols by identifying the range in which you lie in the 0 → 1 space.

Send 'b'



'b' is the range [0.5, 0.75)

Send 'ba'



'ba' is the range [0.5, 0.625)

Eg 4.1

a	0·3
b	0·5
c	0·1
d	0·2

send 'cab'

Squeeze it down to 1 number in that range

'ba': $[0.5, 0.625)$

Idea: Instead of sending two numbers, send the shortest number ('singular)

$[0.5, 0.625) \rightarrow$ Send 0.5 or 0.6

$\rightarrow$ Send 5 or 6 since the 0. is implicit.

Often we use binary to send and this affects our choice.

'ba' $\rightarrow [0.5, 0.625) \rightarrow [0.1_b, 0.101_b)$

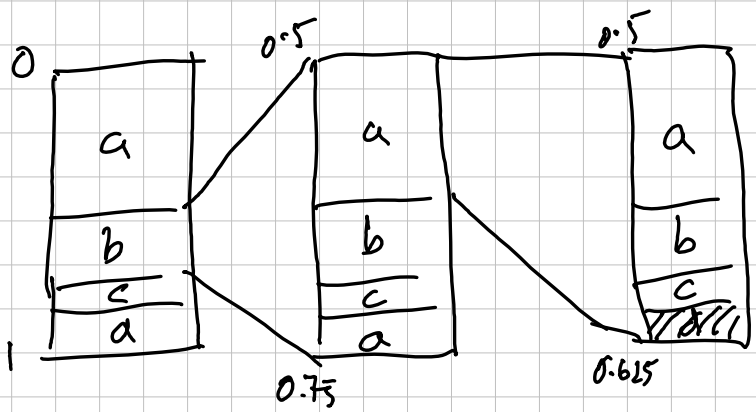
$\rightarrow 0.1_b \rightarrow \underline{\underline{1}}$

Huffman vs Arithmetic

Huffman

'bad' → 100111

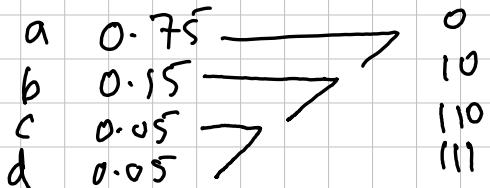
Arithmetic



Decimal: [0.609375, 0.625) → 0.61 → 61

Binary: [0.100111, 0.101) → 0.100111 → 100111

Huffman



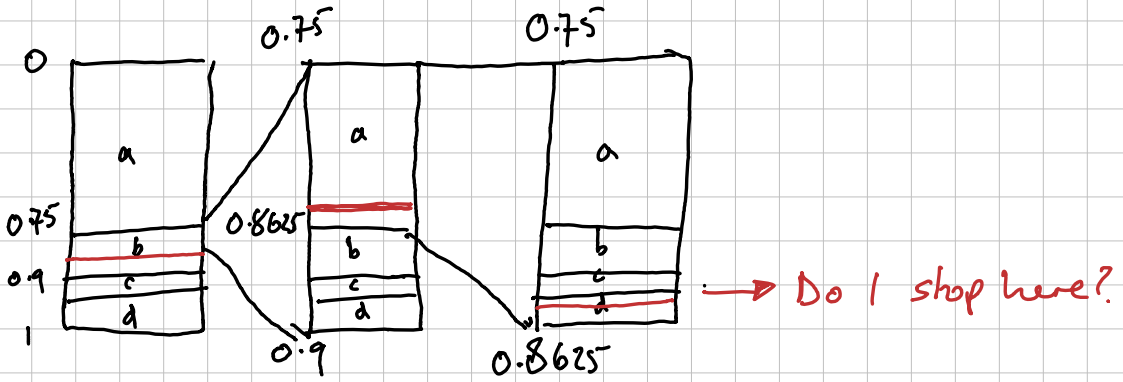
bad $\Rightarrow$ 100111

Eg, 4.2

Decode

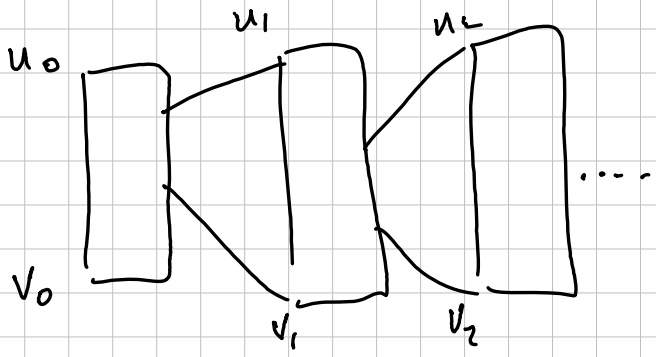
a	0.75
b	0.15
c	0.05
d	0.05

$86 \rightarrow 0.86$



∴ In order to decode, we need to add a terminating character to our message

Formula for limits



$Q(a)$ = cumulative symbol prob. up to symbol a exclusive.

$R(a)$ = Same as $Q(a)$ but inclusive

E.g.

a	0.5	$Q(b) = 0.5$
b	0.2	$R(b) = 0.7$
c	0.3	

$u_0 = 0.0$

$v_0 = 1.0$

$$u_i = u_{i-1} + Q_i(x_i)\left(v_{i-1} - u_{i-1}\right)$$

$$v_i = v_{i-1} + R_i(x_i)\left(v_{i-1} - u_{i-1}\right)$$

a	0.85
b	0.075
#	0.075

Send aaaaaaaaaaaaa a #

$P(a) = 0.85$

$h(a) = 0.231$

x_i	u_i	v_i	H.C.	A.C	Total Info	
a	0	0.85	0	–	0.23	
a	0	0.72	0	–	0.46	
a	0	0.61	0	–	0.70	
a	0	052	0	–	0.94	
a	0	0.44	0	0	1.17	*
a	0	0.37	0	–	1.40	
a	0	0.32	0	–	1.64	
a	0	0.27	0	–	1.88	
a	0	0.23	0	0	2.11	*
a	0	0.19	0	–	2.34	
a	0	0.16	0	–	2.57	
a	0	0.14	0	–	2.81	
a	0	0.1209	0	0	3.05	*
#	$0.85^{13} \times 0.925$	0.85^{13}	11	1111	6.79	

$0.111837\,5815 \equiv 0.0001110011\ldots$

$0.1209\,0549 \equiv 0.0001111011\ldots$

∴ H.C.: 0000000000000011

A.C.: 0001111

So A.C. is not bound to output a codeword per symbol.

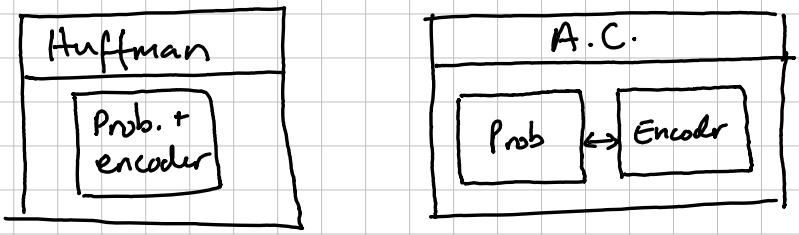
⇒ It outputs a bit for every bit of info we get

Caveat: The # costs us ← Mackay has an analysis

A.C. Flexibility

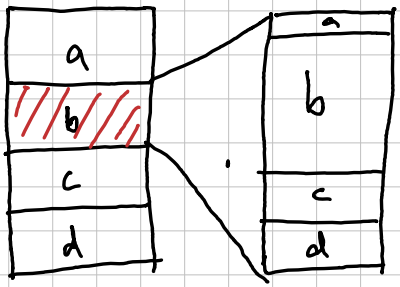
The better we model $P(x)$ the better the compression.

Problem: Most data streams have <u>changing</u> $P(x)$



→ A.C. separates the prob. model from the encoder

→ This allows the probabilities to change during encoding



Requirement: the decoder must be able to produce the exact $P(x)$ sequence!

LLMs for compression

An LLM outputs a prob. dist. for the next token that adapts to the context

← The ideal model for compression!!!

Google Deepmind → "Language Modeling is Compression"

Images compression rate
48%
vs
58.5% (PNG)

Audio comp. rate
21%
vs
30.9% (FLAC)

But you do need both ends to have the LLM access!

Overview

H.C. – Explicit const. prob. model.

A.C. – More advanced works on static and dynamic prob. model.

Next

Lempel-Ziv – Avoid prob. altogether
Memoises sequences + uses pointers back to things it has seen.

Lempel-Ziv

String to send a a b b b c a a

Core idea – remember sequences as you see them + assign codes to them in a dictionary

a a b b b c a a
↑ ↑ ↑ ...
symbol

a a b b b c a a
↑
composite symbols.

Abstractly .... S X ....

S: symbol or composite symbol.

X: symbol

Example $A_x = \{a, b, \#\}$

Compress

String a a a a a b a a a a b a a #

..... SX .....

E.g. 4.2

E.g. 4.3

Dict size?

Why 3 ? Just convenient

⇒ optimize the size based on inputs.

In practice – dynamically resize the dict.

– Somehow tell the decoder that happened

Inefficient

aa
aaa
aaaa ← improbable / never occurs ⇒ LRU

Where is LZ found.

GIF

TIFF

compress

gzip

zip

} DEFLATE algo — LZ encoding, Huffman coding