

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25) Philipps-Universität Marburg version 2024-11-30 23:40 H

Learning Outcomes

Unit 8: Clever Codes

- 1. Know the principles and performance characteristics of *arithmetic coding*.
- **2.** Judge the use of arithmetic coding in applications.
- 3. Understand the context of *error-prone communication*.
- 4. Understand concepts of *error-detecting codes* and *error-correcting codes*.
- 5. Know and understand *Hamming codes*, in particular (7,4) Hamming code.
- 6. Reason about the *suitability of a code* for an application.

Outline

8 Clever Codes

- 8.1 Arithmetic Coding
- 8.2 Practical Arithmetic Coding
- 8.3 Error Correcting Codes
- 8.4 Coding Theory
- 8.5 Hamming Codes

8.1 Arithmetic Coding

Stream Codes

- **Recall:** (binary) character encoding $E : \Sigma \to \{0, 1\}^*$
 - Huffman codes *optimal* for any given character frequencies
 - $\rightsquigarrow\,$ encoding all characters with that code minimizes compressed size
 - ... if we assume that all characters must be encoded individually by a codeword!
- Stream codes instead compress entire **sequence** of characters
 - ▶ RLE and LZW are examples of stream codes → can sometimes do better
- Two indicative examples
 - **1.** "Low entropy bits:" $\Sigma = \{0, 1\}$, highly skewed: $p_0 = 0.99$
 - → entropy $\mathcal{H}(\frac{1}{100}, \frac{99}{100}) \approx 0.08$ bits per character, Huffman code must use 1 bit per character!
 - → "optimal" Huffman code gives 12-fold space increase over entropy!
 - Can certainly do better here (RLE!)
 - **2.** "Trits": $\Sigma = \{0, 1, 2\}$, equally likely
 - → entropy $\mathcal{H}(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}) = \lg(3) \approx 1.58$ bits per character, Huffman code uses average of $\frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3} \approx 1.67$

Can we do better?

A Decent Hack: Block Codes

- Huffman on trits wastes ≈ 0.0817 bits per character and over 5 % of space
- A simple trick can reduce this substantially!
 - treat 5 trits as one "supercharacter", e.g., 21101
 - $\rightsquigarrow 3^5 = 243$ possible combinations
 - \rightarrow encode these using 8 bits (with $2^8 = 256$ possible combinations)
 - entropy $lg(3^5) \approx 7.92$ bits, so less than 0.1% wasted space!

▶ We can even use a Huffman code for the supercharacters to handle nonuniformity!

- ▶ For the low-entropy bits, could use 3 bits
 - \rightsquigarrow probabilities:

000 : 0.97 001, 010, 100 : 0.0098 011, 101, 110 : 0.000099 111: 0.000001

- \rightsquigarrow with Huffman code, 1.06 bits per superchar of 3 input bits
- → almost factor 3 better; can improve with larger blocks!

Block Codes – A Panacea?

Using supercharacters works well in our examples.



Hmmm ... so why don't we treat the entire source text as one large block? Wouldn't that be even better!?

- → We can optimally compress any text, without doing anything intelligent!?
- **F** For general case, need to *communicate* the supercharacter encoding
 - Blocks of *k* characters need $\Omega(\sigma^k)$ space for code
 - Huffman code has to be part of coded message
 - $\rightsquigarrow\,$ Can only sensibly use block codes for small σ and k



There is no such thing as a free lunch . . .



Arithmetic Coding

Also: Block codes still had $\Theta(n)$ wasted space for sequences of *n* symbols

► Arithmetic Coding:

- **0.** Maintain $[\ell, \ell + p) \subseteq [0, 1)$; initially $\ell = 0, p = 1$
- 1. Zoom into subinterval for each character
- 2. Output dyadic encoding of final interval

► *Step 1:* "Zoom" for each character (trit) in *S*[0..*n*):

- ► Of the current subinterval $[\ell, \ell + p)$, take first, second or last third depending whether S[i] = 0, 1, resp. 2: $\ell := \ell + S[i] \cdot \frac{1}{3} \cdot p$ $p := p \cdot \frac{1}{3}$
- ► *Step 2:* Dyadic encoding
 - Find smallest *m* so that $\exists x \in \mathbb{N}_0$ with $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right] \subseteq [\ell, \ell+p)$
 - Output *x* in binary using *m* bits.
- \rightsquigarrow Encode *n* trits in $n \lg(3) + 2$ bits(!) without cheating



Arithmetic Coding – Encode Trits Example

- \blacktriangleright S[0..n) = 21101 (n = 5)
- **Step 1:** Zoom into subintervals

Iteration	l	р	Interval (rounded)	
0	0	1	[0.00000, 1.00000)	I
1	$\frac{2}{3}$	$\frac{1}{3}$	[0.66667, 1.00000)	H
2	$\frac{7}{9}$	$\frac{1}{9}$	[0.77778, 0.88889)	
3	$\frac{22}{27}$	$\frac{1}{27}$	[0.81482, 0.85185)	н
4	$\frac{66}{81}$	$\frac{1}{81}$	[0.81482, 0.82716)	Н
5	$\frac{199}{243}$	$\frac{1}{243}$	[0.81893, 0.82305)	1

• **Step 2:** Dyadic encoding for interval $[\ell, \ell + p) = \left[\frac{199}{243}, \frac{200}{243}\right]$

- Must have $m \ge \lg(1/p) > 7$
 - ▶ m = 8: smallest $x/2^m \ge \frac{199}{243}$ is x = 210, but $[210/256, 211/256) \approx [0.82031, 0.82422) \notin [\ell, \ell + p]$
 - ▶ m = 9: smallest $x/2^m \ge \frac{199}{243}$ is x = 420 and $[420/512, 421/512) \approx [0.82031, 0.82227) \subset [\ell, \ell + p)$

 \rightarrow Output *x* = 420 in binary with *m* = 9 digits: 110100100

Versatility of Arithmetic Coding – Adaptive Model



Arithmetic Coding – General framework

- Note: Arithmetic coder *doesn't care* if probabilities or even σ change all the time!
 - As long as encoder and decoder know from context what they are!

General stochastic sequence:

Sequence of random variables X_0, X_1, X_2, \ldots such that

- **1.** $X_i \in [0..U_i) \cup \{\$\}$ (We use \$ to signal "end of text")
- **2.** $\mathbb{P}[X_i = j] = P_{ij}$
- 3. both U_i and P_{ij} are random variables as they *depend* on X_0, \ldots, X_{i-1} , but conditioned on X_0, \ldots, X_{i-1} , they are fixed and known: $P_{ij} = P_{ij}(X_0, \ldots, X_{i-1}) = \mathbb{P}[X_i = j | X_0, \ldots, X_{i-1}]$ $U_i = U_i(X_0, \ldots, X_{i-1}) = \max\{j : P_{ij} > 0\}$
- Can model arbitrary dependencies on previous outcomes
- Assume here that random process is known by both encoder and decoder (fixed coding) otherwise extra space needed to encode model!

Arithmetic Coding – Encoding

¹ **procedure** arithmeticEncode(X_0, \ldots, X_n): *// Assume model* U_i *and* P_{ij} *are fixed.* 2 // Assume $X_i \in [0..U_i)$ for i < n and $X_n =$ 3 // Step 1: Interval zooming 4 $\ell := 0; p := 1$ 5 **for** i := 0, ..., n - 1 **do** 6 $X_i - 1$ $q := \sum_{i=0}^{N_l} P_{ij};$ 7 $\ell := \ell + q \cdot p; \quad p := p \cdot P_{i,X_i}$ 8 end for 9 $q := 1 - P_{n,\$}$ // encode \$ as last character 10 $\ell := \ell + q \cdot p; \quad p := p \cdot P_{n,\$}$ 11 // Step 2: Dyadic encoding 12 $m := \lceil \lg(1/p) \rceil - 1$ 13 do 14 $m := m + 1; \ x := \lceil \ell \cdot 2^m \rceil$ 15 while $(x + 1)/2^m > \ell + p$ 16 **return** *x* **in** binary using *m* bits 17

Arithmetic Coding – Decoding

1	procedure arithmeticDecode(<i>C</i> [0 <i>m</i>)):						
2	// Assume model U_i and P_{ij} are fixed.						
3	// C[0m) bit string produced by arithmeticEncode						
4	$x = \sum_{i=0}^{m} C[i] \cdot 2^{-(i+1)} // final interval [x/2^{m}, (x+1)/2^{m})$						
5	$\ell := 0; p := 1; i := 0$						
6	while true						
7	c := 0; q := 0 // Decode next character c						
8	while $\ell + q \cdot p < x/2^m$						
9	if $c == U_i + 1 // reached $$						
10	X[i] := \$						
11	return X[0 <i>i</i>]						
12	else						
13	$q := q + P_{i,c}; \ c := c + 1$						
14	end while						
15	$c := c - 1; q := q - P_{i,c} // we overshot by 1$						
16	X[i] := c						
17	$\ell := \ell + q \cdot p; p := p \cdot P_{i,c}$						
18	i := i + 1						
19	end for						

8.2 Practical Arithmetic Coding

Arithmetic Coding – Numerics

- As implemented above, *p* usually gets smaller by a constant factor with *each character* ~ *p* gets exponentially small in *n*!
 - *l* does not get smaller in absolute terms, but we need it to ever higher accuracy
- \rightsquigarrow requires $\Omega(n)$ bit precision and exact arithmetic!
- With a clever trick, this can be avoided!
 - If $[\ell, \ell + p) \subseteq [0, \frac{1}{2})$, we know:
 - Our final x with $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right] \subseteq [\ell, \ell+p)$ must start with a 0-bit!
 - → Output a 0 and renormalize interval: $\ell := 2\ell; p := 2p$
 - If $[\ell, \ell + p) \subseteq [\frac{1}{2}, 1)$, similarly:
 - Output 1 and renormalize: $\ell := \ell - \frac{1}{2}$ $\ell := 2\ell; p := 2p$

	- 00000
	- 00000 0000
	- 00010 000
	- 00011 0001
	00100 00
	- 00101 0010
	- 00110 001
a	= 00111 0011
	- 01000 0
	- 01001 0100
	- 01010 010
	= 01011 0101
	= 01100 0110
	- 01101 0110
	- 01110 0111
ba	- 01111 0111
	= 10000 1000
11.	- 10001 1000
DDa	= 10010 1001
<u>bbba</u>	- 10011 1001 10
b	- 10100 1010
op ppp ppp	- 10101 101
	= 10110 1011
bbb□	- 10111 1
bb□	- 11000 1100
b	11001 110
	<u> </u>
	- 11011 11
	11100 1110
	- 11110 111
	- 11111 1111

Arithmetic Coding – Renormalization

Does this guarantee l *and* p *stay in a reasonable range?*

No! Consider (uniform) trits in {0, 1, 2} again and encode 111111111111111...

$$\implies p = \left(\frac{1}{3}\right)^n, \quad \ell = \frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \dots = \sum_{i=1}^n 3^{-i} = \frac{1}{2} - \frac{3^{-n}}{2} \\ \implies \ell < \frac{1}{2} \text{ and } \ell + p > \frac{1}{2} \implies \text{next bit unknown as of yet}$$

But: If $[\ell, \ell + p) \subseteq [\frac{1}{4}, \frac{3}{4})$, next **two** bits are either 01 or 10

- Remember an "outstanding opposite bit" (increment counter)
- Renormalize:

 $\ell := \ell - \frac{1}{4}$ $\ell := 2\ell; \ p := 2p$

- $\rightsquigarrow \ell$ and *p* remain in range of *P*_{*ij*}
- \rightsquigarrow round P_{ij} to integer multiple of $2^{-F} \rightsquigarrow$ fixed-precision arithmetic

		_
<u>- 000</u> 00 0000		
<u>= 00001</u> 000		
= 00010 0001		
- 00011	00	
= 00100 = 00100 0010		
= 00101 001		
= 00110 = 00110 0011		
- 00111		0
= 01000 0100		
- 01001 010		
$\frac{-01010}{-01011}$ 0101		
- 01011	01	
= 01100 0110		
= 01101 011		
= 01110 0111		
- 10000		-
= 10001 ¹⁰⁰⁰		
= 10010 1001		
= 10011 1001	10	
- 10100	10	
- 10101 1010		
$= 10110_{1011}$		
- 10111 1011		1
$\frac{-11000}{1100}$		-
$= 11001^{1100}$ 110		
= 11010 1101		
- 11011	11	
$\frac{-11100}{-1110}$ 1110		
- 11101 111		
$\frac{-11110}{-1111}$ 1111		
- 11111		

Fixed Precision Arithmetic Encode

Detailed code from Moffat, Neal, Witten, Arithmetic Coding Revisited, ACM Trans. Inf. Sys. 1998

```
arithmetic_encode(l, h, t)
```

/* Arithmetically encode the range [l/t, h/t) using low-precision arithmetic. The state variables R and L are modified to reflect the new range, and then renormalized to restore the initial and final invariants $2^{b-2} < R \leq 2^{b-1}$, $0 \leq L < 2^{b} - 2^{b-2}$, and $L + R \leq 2^{b} */$

- (1) Set $r \leftarrow R \operatorname{div} t$
- (2) Set $L \leftarrow L + r$ times l
- (3) If h < t then

set $R \leftarrow r$ times (h - l)

else

set $R \leftarrow R - r$ times l

(4) While $R \leq 2^{b-2}$ do

Use Algorithm ENCODER RENORMALIZATION (Figure 7) to renormalize R, adjust L, and output one bit

Fixed Precision Renormalize

In arithmetic_encode() /* Reestablish the invariant on R, namely that $2^{b-2} < R \le 2^{b-1}$. Each doubling of R corresponds to the output of one bit, either of known value, or of value opposite to the value of the next bit actually output */ (4) While $R \le 2^{b-2}$ do If $L + R \le 2^{b-1}$ then bit_plus_follow(0) else if $2^{b-1} \le L$ then bit_plus_follow(1) Set $L \leftarrow L - 2^{b-1}$ else Set bits_outstanding \leftarrow bits_outstanding + 1 and $L \leftarrow L - 2^{b-2}$ Set $L \leftarrow 2L$ and $R \leftarrow 2R$

 $bit_plus_follow(x)$

/* Write the bit x (value 0 or 1) to the output bit stream, plus any outstanding following bits, which are known to be of opposite polarity */

(1) $write_one_bit(x)$.

```
(2) While bits_outstanding > 0 do
```

```
write_one_bit(1 - x)
Set bits_outstanding \leftarrow bits_outstanding -1
```

Fixed Precision Arithmetic Decode

 $decode_target(t)$

/* Returns an integer target, $0 \le target < t$ that is guaranteed to lie in the range [l, h) that was used at the corresponding call to arithmetic_encode() */

- (1) Set $r \leftarrow R$ div t
- (2) Return $(\min\{t-1, D \operatorname{div} r\})$

```
arithmetic_decode(l, h, t)
```

/* Adjusts the decoder's state variables R and D to reflect the changes made in the encoder during the corresponding call to *arithmetic_encode()*. Note that, compared with Algorithm CACM CODER (Figure 6), the transformation D = V - L is used. It is also assumed that r has been set by a prior call to *decode_target()* */

- (1) Set $D \leftarrow D r$ times l
- (2) If h < t then

```
set R \leftarrow r times (h - l)
```

else

```
(3) While R \leq 2^{b-2} do
Set R \leftarrow 2R and D \leftarrow 2D + read_one_bit()
```

Arithmetic Coding Discussion

- C Subtle code
- Typically slower to encode/decode than Huffman codes
- D Encoded bits produced/consumed in bursts
- Extremely versatile w. r. t. random process
- Almost optimal space usage / compression
- Widely used (instead of Huffman) in JPEG, zip variants, ...

8.3 Error Correcting Codes

Noisy Communication

- most forms of communication are "noisy"
 - humans: acoustic noise, unclear pronunciation, misunderstanding, foreign languages
- How do humans cope with that?
 - slow down and/or speak up
 - ask to repeat if necessary
- But how is it possible (for us) to decode a message in the presence of noise & errors?

Bcaesue it semes taht ntaurul lanaguge has a lots fo **redundancy** bilt itno it!

- \rightsquigarrow We can
- **1. detect errors** "This sentence has aao pi dgsdho gioasghds."
- 2. correct (some) errors "Tiny errs ar corrrected automaticly." (sometimes too eagerly as in the Chinese Whispers / Telephone)



Noisy Channels

- computers: copper cables & electromagnetic interference
- transmit a binary string
- but occasionally bits can "flip"
- \rightsquigarrow want a robust code



We can aim at

- **1. error detection** \rightsquigarrow can request a re-transmit
- **2.** error correction \rightarrow
 - \rightsquigarrow avoid re-transmit for common types of errors
- This will require *redundancy*: sending *more* bits than plain message
 goal: robust code with lowest redundancy
 that's the opposite of compression!

8.4 Coding Theory

Block codes

model:

- ▶ want to send message $S \in \{0, 1\}^*$ (bitstream) across a (*communication*) channel
- any bit transmitted through the channel might *flip* (0 → 1 resp. 1 → 0) no other errors occur (no bits lost, duplicated, inserted, etc.)
- ► instead of S, we send encoded bitstream C ∈ {0, 1}* sender encodes S to C, receiver decodes C to S (hopefully)
- $\rightsquigarrow\,$ what errors can be detected and/or corrected?
- all codes discussed here are block codes
 - divide *S* into messages $m \in \{0, 1\}^k$ of *k* bits each $(k = message \ length)$
 - encode each message (separately) as $C(m) \in \{0, 1\}^n$ $(n = block length, n \ge k)$
 - $\rightsquigarrow\ can analyze everything block-wise$
- between 0 and n bits might be flipped invalid code
 - how many flipped bits can we definitely detect?
 - how many flipped bits can we correct without retransmit?

i.e. decoding m still possible

Code distance

▶ each block code is an *injective* function $C : \{0, 1\}^k \to \{0, 1\}^n$

• define
$$C$$
 = set of all codewords = $C(\{0, 1\}^k)$

- $|\mathcal{C}| = 2^k$ out of 2^n *n*-bit strings are valid codewords $\rightsquigarrow \mathcal{C} \subseteq \{0,1\}^n$
- decoding = finding closest valid codeword

distance of code:

 $d = \text{minimal Hamming distance of any two codewords} = \min_{x,y \in \mathcal{C}} d_H(x, y)$

Implications for codes

- **1.** Need distance *d* to **detect** all errors flipping up to d 1 bits.
- 2. Need distance *d* to correct all errors flipping up to $\left|\frac{d-1}{2}\right|$ bits.

Lower Bounds

- Main advantage of concept of code distance: can *prove* lower bounds on block length
 Given block length *n*, message length *k*, code distance *d*, we must have:
- Singleton bound: $2^k \le 2^{n-(d-1)} \iff n \ge k+d-1$

• Hamming bound:
$$2^k \leq \frac{2^n}{\sum_{f=0}^{\lfloor (d-1)/2 \rfloor} {n \choose f}}$$

▶ proof idea: consider "balls" of bitstrings around codewords count bitstrings with Hamming-distance ≤ t = [(d - 1)/2] correcting t errors means all these balls are disjoint so 2^k · ball size ≤ 2ⁿ

 \rightsquigarrow We will come back to these.

8.5 Hamming Codes

Parity Bit

simplest possible error-detecting code: add a parity bit



- \rightsquigarrow code distance 2
- can detect any single-bit error (actually, any odd number of flipped bits)
- used in many hardware (communication) protocols
 - PCI buses, serial buses
 - caches
 - early forms of main memory
- very simple and cheap

Error-correcting codes

any downtime is expensive!

- typical application: heavy-duty server RAM
 - bits can randomly flip (e.g., by cosmic rays)
 - individually very unlikely, but in always-on server with lots of RAM, it happens!

https://blogs.oracle.com/linux/attack-of-the-cosmic-rays-v2





Can we **correct** a bit error without knowing where it occurred? How?

- ► Yes! store every bit *three times*!
 - upon read, do majority vote
 - ▶ if only one bit flipped, the other two (correct) will still win
 - *triples* the cost!



You want WHAT !?!



instead of 200% (!)

Can do it with 11% extra memory!

How to locate errors?

- ► Idea: Use several parity bits
 - each covers a subset of bits
 - clever subsets ~> violated/valid parity bit pattern narrows down error
 - flipped bit can be one of the parity bits!

• Consider n = 7 bits B_1, \ldots, B_7 with the following constraints:



Observe:

- ► No error (all 7 bits correct) $\rightsquigarrow C = C_2 C_1 C_0 = 000_2 = 0$
- ▶ What happens if (exactly) 1 bit, say *B_i* flips?

 $C_j = 1$ iff *j*th bit in binary representation of *i* is 1 \rightarrow *C* encodes position of error!

(7, 4) Hamming Code

► How can we turn this into a code?



▶ B_4 , B_2 and B_1 occur only in one constraint each \rightarrow **define** them based on rest!

- ▶ (7,4) *Hamming Code* Encoding
 - **1.** Given: message $D_3D_2D_1D_0$ of length k = 4
 - **2.** copy $D_3D_2D_1D_0$ to $B_7B_6B_5B_3$
 - **3.** compute $P_2P_1P_0 = B_4B_2B_1$ so that C = 0
 - **4.** send $D_3 D_2 D_1 P_2 D_0 P_1 P_0$

(7, 4) Hamming Code – Decoding

- ► (7,4) *Hamming Code* Decoding
 - **1.** Given: block $B_7B_6B_5B_4B_3B_2B_1$ of length n = 7
 - **2.** compute *C* (as above)
 - 3. if C = 0 no (detectable) error occurred otherwise, flip B_C (the Cth bit was twisted)
 - **4.** return 4-bit message $B_7B_6B_5B_3$

(7, 4) Hamming Code – Properties

Hamming bound:

- 2⁴ valid 7-bit codewords (on per message)
- any of the 7 single-bit errors corrected towards valid codeword
- $\rightsquigarrow~$ each codeword covers 8 of all possible 7-bit strings
- $2^4 \cdot 2^3 = 2^7 \quad \rightsquigarrow \quad \text{exactly cover space of 7-bit strings}$
- distance d = 3
- can *correct* any 1-bit error
- How about 2-bit errors?
 - We can *detect* that *something* went wrong.
 - ▶ But: above decoder mistakes it for a (different!) 1-bit error and "corrects" that
 - ► Variant: store one additional parity bit for entire block
 - → Can *detect* any 2-bit error, but *not correct* it.

Hamming Codes – General recipe

- construction can be generalized:
 - Start with $n = 2^{\ell} 1$ bits for $\ell \in \mathbb{N}$ (we had $\ell = 3$)
 - use the ℓ bits whose index is a power of 2 as parity bits
 - the other $n \ell$ are data bits
- Choosing l = 7 we can encode entire word of memory (64 bit) with 11% overhead (using only 64 out of the 120 possible data bits)

simple and efficient coding / decoding
fairly space-efficient

Outlook

► Indeed: $(2^{\ell}-1, 2^{\ell}-\ell-1)$ Hamming Code is "*perfect*" code

 \rightsquigarrow cannot use fewer bits . . .

= matches Hamming lower bound

- if message length is 2^ℓ ℓ 1 for ℓ ∈ N≥2
 i. e., one of 1, 4, 11, 26, 57, 120, 247, 502, 1013, ...
- and we want to correct 1-bit errors

▶ For other scenarios, finding good codes is an active research area

- ▶ information theory predicts that *almost all* randomly chosen codes are good(!)
- but these are inefficient to decode
- $\rightsquigarrow~$ clever tricks and constructions needed