

# Data compression

## Probabilistic models

I. Holmes

Department of Bioengineering  
University of California, Berkeley

Spring semester

# Outline

- 1 Information Theory and Compression
- 2 Theoretical Limits of Compression
- 3 Practical Compression Schemes
- 4 Compression of Biological Sequences

## Information-theoretic quantities

Shannon information:  $h(x) = -\log_2 p(x)$

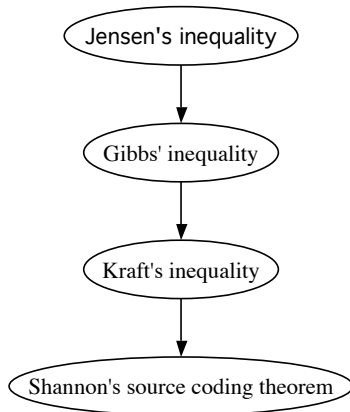
Shannon entropy:  $H_p = \langle h(x) \rangle_p$

Relative entropy:  $D(p||q) = \langle \log \frac{p(x)}{q(x)} \rangle_p$

Mutual information:  $I(x; y) = D(p(x, y)||p(x)p(y))$

Kolmogorov complexity:  $K(\mathbf{x}) = \text{size of smallest program that outputs } \mathbf{x}$

# Theorems



**Information Theory, Inference and Learning Algorithms.**  
MacKay (2003)

# Jensen's Inequality

- A function  $f(x)$  is **convex** over  $(a, b)$  if every chord lies above the function.
  - $f''(x) \geq 0$  is sufficient.
- Jensen's inequality:

$\text{If } x \text{ is an rv and } f(x) \text{ is convex, then } \langle f(x) \rangle \geq f(\langle x \rangle)$
- If  $f(x)$  is concave, then inequality is reversed.

# Jensen's Inequality

If  $x$  is an rv and  $f(x)$  is convex, then  $\langle f(x) \rangle \geq f(\langle x \rangle)$

- Physical version:
  - Let  $p(x)$  be  $x$ -distribution of masses on curve  $y = f(x)$ .
  - Then center of gravity  $(\langle x \rangle, \langle f(x) \rangle)$  lies above curve.

# Bounds on the Shannon entropy

$$H_p \leq \log |\Omega|$$

Proof: apply Jensen's inequality to  $u = 1/p(x)$ ,  $f(u) = -\log u$

$$\langle f(u) \rangle \geq f(\langle u \rangle)$$

# Gibbs' Inequality

$$D(p||q) \geq 0 \quad \text{with equality iff } p = q$$

where  $p, q$  are probability distributions over the same set  $\Omega$ , and

$$D(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

is the **relative entropy** or **Kullback-Leibler divergence**.

Proof: apply Jensen to  $u = q(x)/p(x)$ ,  $f(u) = -\log u$

$$\langle f(u) \rangle \geq f(\langle u \rangle)$$



# Symbol codes

- Let  $\Omega^+$  be the set of all finite-length strings over  $\Omega$
- A **binary symbol code**  $C$  is a mapping from  $\Omega$  to  $\{0, 1\}^+$
- The **extended code**  $C^+$  is a mapping from  $\Omega^+$  to  $\{0, 1\}^+$  obtained by concatenating codewords:

$$C^+(x_1 x_2 \dots x_N) = C(x_1)C(x_2) \dots C(x_N)$$

- Side note: it's common to add an **End Of File** symbol to the alphabet, so the decoder knows when to stop
  - although it's usually more efficient to send the length of your message first, followed by the symbols

# Decodability

- The code  $C$  is **uniquely decodable** if, under the extended code  $C^+$ , no two strings have the same encoding.
- If no codeword is a prefix of any other codeword, then  $C$  is a **prefix code**.
- All prefix codes are uniquely decodable...

# Block codes

- A **block code** is a code over the alphabet  $\Omega^N$
- That is, it divides an  $\Omega$ -string into blocks of size  $N$
- A common efficiency trick in information theory
  - e.g.  $|\Omega| = 5$ , uniform: Shannon info  $\sim 2.32$  bits/symbol
  - Naive encoding would require 3 bits/symbol: 0.7 bits/symbol wasted
  - Block code with  $N = 3$  requires  $7/3 \simeq 2.33$  bits/symbol
- For any symbol code  $C$  there is a trivial block code  $C^N$

- Let  $l(x)$  be the length of the codeword  $C(x)$
- Let  $p(x)$  be the probability distribution over symbols
- Define the *expected codeword length* of  $C$ :

$$L_p(C) = \langle l(x) \rangle_p$$

- **Shannon's Source Coding Theorem** states that

$$L_p(C) \geq H_p$$

for any uniquely decodable code  $C$ .

(Technically, this is the *converse* of the SCT; the SCT itself states that the equality is asymptotically achievable.)

- The equality holds if  $l(x) = h(x) \quad \forall x$

# Insight into Source Coding Theorem

- Given a unique code  $C$ , is it optimal for anything?
- Maybe... if there's a distribution  $q(x)$  such that  $L_q(C) = H_q$
- This would require that  $l(x) = h_q(x) \quad \forall x$ , i.e.

$$q(x) = 2^{-l(x)} / z$$

where

$$z = \sum_{x \in \Omega} 2^{-l(x)}$$

is a **partition function**

- We use these “implicit probabilities” to prove the inequality
  - Note that  $l(x) = \log(1/q(x)) - \log z$

## Partition function for block code

- Partition function for symbol code  $C$  is

$$z = \sum_{x \in \Omega} 2^{-l(x)}$$

- Partition function for block code  $C^N$  is

$$\begin{aligned} z^N &= \left[ \sum_{x \in \Omega} 2^{-l(x)} \right]^N \\ &= \sum_{x_1 \in \Omega} \sum_{x_2 \in \Omega} \cdots \sum_{x_N \in \Omega} 2^{-(l(x_1) + l(x_2) + \dots + l(x_N))} \end{aligned}$$

## Rearranging the partition function

$$z^N = \sum_{x_1 \in \Omega} \sum_{x_2 \in \Omega} \cdots \sum_{x_N \in \Omega} 2^{-(I(x_1) + I(x_2) + \dots + I(x_N))}$$

- The quantity  $I(x_1) + I(x_2) + \dots + I(x_N)$  is the length of the  $C^+$ -encoding of the string  $\mathbf{x} = x_1 x_2 \cdots x_N$ .
- For every string  $\mathbf{x} \in \Omega^N$ , there is one term in the above sum for  $z^N$ .
- Let
  - $A_l$  be the number of strings  $\mathbf{x}$  having encoded length  $l$
  - $l_{\min} = \min_{\mathbf{x}} I(\mathbf{x})$
  - $l_{\max} = \max_{\mathbf{x}} I(\mathbf{x})$

$$z^N = \sum_{l=l_{\min}}^{l_{\max}} 2^{-l} A_l$$

# The Kraft-McMillan inequality

- $A_l$  is number of strings with encoded length  $l$ :

$$z^N = \sum_{l=Nl_{\min}}^{Nl_{\max}} 2^{-l} A_l$$

- If  $C$  is uniquely decodable, then  $A_l \leq 2^l$ . Therefore

$$z^N = \sum_{l=Nl_{\min}}^{Nl_{\max}} 2^{-l} A_l \leq \sum_{l=Nl_{\min}}^{Nl_{\max}} 1 \leq Nl_{\max}$$

- Since this holds for arbitrarily large  $N$ , it must be that

$$z \leq 1$$

- This is the **Kraft-McMillan inequality**, now for the source coding theorem



# Proof of (converse) Source Coding Theorem

- Recall that

$$L_p(C) = \langle I(x) \rangle_p \quad (\text{definition of } L_p)$$

$$I(x) = \log(1/q(x)) - \log z \quad (\text{definition of } q)$$

$$\langle \log 1/q(x) \rangle_p \geq \langle \log 1/p(x) \rangle_p \quad (\text{Gibbs})$$

$$z \leq 1 \quad (\text{Kraft-McMillan})$$

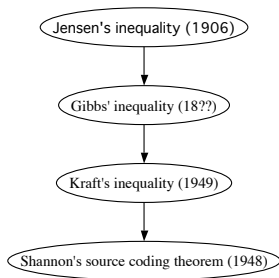
- It follows that

$$L_p(C) = \langle \log 1/q(x) \rangle_p - \log z$$

$$\geq \langle \log 1/p(x) \rangle_p - \log z$$

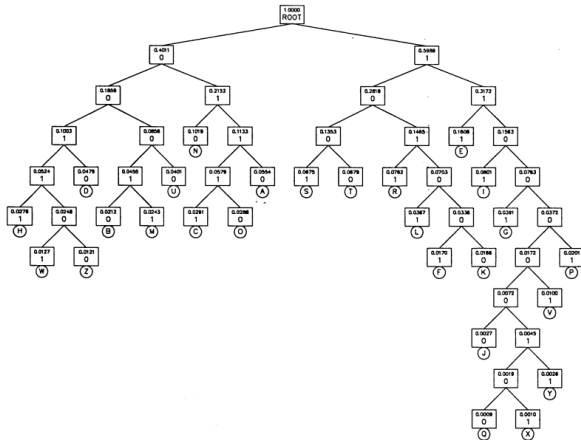
$$\geq H_p$$

## Theorems (with dates)

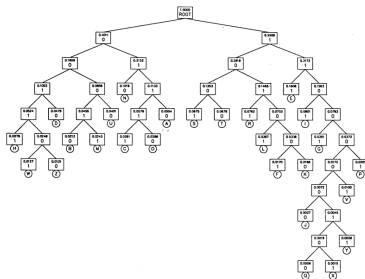


- Gibbs worked extensively on statistical mechanics.  
**Where/when was Gibbs' inequality published?**
- McMillan independently rediscovered Kraft's result in 1956.
- Kraft & McMillan cited (respectively) Redheffer & Doob.
- Shannon's formulation of his theorem was slightly different (asymptotic error rate of general block code).

# Huffman coding

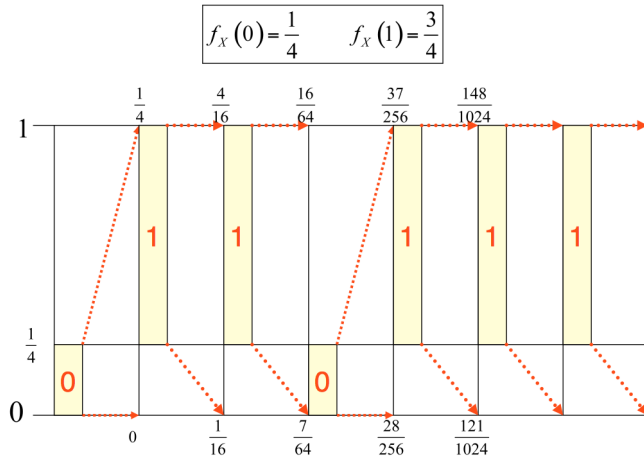


# Huffman coding algorithm



- **Huffman tree** generates optimal prefix codes with  $L_p(C) < H_p + 1$  (but NB overhead of 1 bit/symbol).
  - Bottom-up tree construction algorithm: combine the two least frequent symbols into a single symbol, and repeat.

# Arithmetic coding



# Arithmetic coding

- **Arithmetic coding** is a block (not symbol) compression scheme that achieves asymptotically perfect efficiency.
- The message  $\mathbf{x}$  is represented as a subinterval of  $[0, 1)$ 
  - Subinterval found by recursive subdivision
  - Width of subinterval  $\simeq$  probability of message  $P(\mathbf{x})$
- Represent message as shortest binary fraction in subinterval
  - Guaranteed to be expressible in  $\sim -\log_2 P(\mathbf{x})$  bits

## Arithmetic coding: the adaptive model

- Represent the alphabet numerically:  $\Omega = \{1, 2 \dots N\}$
- Suppose we have encoded  $k - 1$  characters of message  $\mathbf{x}$
- Let  $k - 1$ 'th interval be  $[a, a + b)$
- Next ( $k$ 'th) interval is  $[a', a' + b')$  where

$$a' = a + b \sum_{\omega=1}^{x_k-1} p_k(\omega)$$

$$b' = b p_k(x_k)$$

$$p_k(\omega) = P(x_k = \omega | x_1 \dots x_{k-1})$$

- Note that  $p_k$  is **adaptive** — it has a “memory” of the last  $k - 1$  encoded/decoded characters

# Arithmetic coding: issues

- **Underflow #1:** finite precision
  - Range delimiters  $[a, b)$  may reach hardware precision limit
  - Solution: while  $a$  and  $b$  have same top digit: output, shift, add padding digits
    - e.g.  $[\text{.436}, \text{.437}) \rightarrow [\text{.600}, \text{.799})$  and output “43”
    - NB we pad  $a$  with 0's and  $b$  with 9's, to maximize range
- **Underflow #2:** delimiters straddle boundary
  - Decimal equivalent: ranges like  $[\text{0.499999995}, \text{0.500000005})$
  - Top digits don't converge before underflow occurs
  - Solution: extract next most significant digits, output them later (when top digits converge)
- **Floating-point math:** don't trust reproducibility of FPU
  - No room for vagaries of floating-point math
  - Solution: truncate all probabilities to some finite fraction
  - Use integers throughout code (e.g. discretized log-probs)



## Arithmetic coding: issues

- **Underflow #1:** finite precision
  - Range delimiters  $[a, b)$  may reach hardware precision limit
  - Solution: while  $a$  and  $b$  have same top digit: output, shift, add padding digits
    - e.g.  $[\text{.436}, \text{.437}) \rightarrow [\text{.600}, \text{.799})$  and output “43”
    - NB we pad  $a$  with 0's and  $b$  with 9's, to maximize range
- **Underflow #2:** delimiters straddle boundary
  - Decimal equivalent: ranges like  $[0.49999995, 0.50000005)$
  - Top digits don't converge before underflow occurs
  - Solution: extract next most significant digits, output them later (when top digits converge)
- **Floating-point math:** don't trust reproducibility of FPU
  - No room for vagaries of floating-point math
  - Solution: truncate all probabilities to some finite fraction
  - Use integers throughout code (e.g. discretized log-probs)

## Arithmetic coding: issues

- **Underflow #1:** finite precision
  - Range delimiters  $[a, b)$  may reach hardware precision limit
  - Solution: while  $a$  and  $b$  have same top digit: output, shift, add padding digits
    - e.g.  $[\text{.436}, \text{.437}) \rightarrow [\text{.600}, \text{.799})$  and output “43”
    - NB we pad  $a$  with 0's and  $b$  with 9's, to maximize range
- **Underflow #2:** delimiters straddle boundary
  - Decimal equivalent: ranges like  $[0.49999995, 0.50000005)$
  - Top digits don't converge before underflow occurs
  - Solution: extract next most significant digits, output them later (when top digits converge)
- **Floating-point math:** don't trust reproducibility of FPU
  - No room for vagaries of floating-point math
  - Solution: truncate all probabilities to some finite fraction
  - Use integers throughout code (e.g. discretized log-probs)

## Other symbol and number codes

- Encoding symbols from finite alphabets
  - **Binary encoding**: use  $k = \log_2 n$  bits to encode  $n$  symbols
  - **Truncated binary encoding**: let  $b = n - 2^k$ .
    - First  $2^k - b$  symbols use first  $2^k - b$  codewords of length  $k$
    - Remaining  $2b$  symbols use **last** codewords of length  $k + 1$
- Encoding arbitrary integers,  $n$ 
  - **Unary encoding**: transmit number  $n$  as  $n$  1's, followed by a 0
  - **Golomb encoding**: choose parameter  $m$ 
    - transmit  $n/m$  using unary encoding
    - transmit  $n \% m$  using truncated binary encoding

Each of these is optimal for some probability distribution:

can you see which?

## Codes based on repetition

- Each of these has a shorthand for some repetition
  - **Run-length encoding**: repeated characters
  - Dictionary (**Lempel-Ziv**) encoding: repeated words
    - LZ77: sliding window
    - LZ78: adaptive dictionary
- **Which probability distribution is each optimal for?**
  - Imagine a process that generates such patterns....

# Probabilistic models used with arithmetic coding

- “Adaptive arithmetic coding”: updated frequency tables
- Order- $N$  Markov chain
- **PPM**: Prediction by Partial Matching
  - Attempts to predict using the previous  $n$  characters
  - If previous  $n$  not seen before, drops to  $n - 1$ , then  $n - 2$ , etc.
  - If no context seen, uses a flat (or, in PPMd, adaptive) prior
- **PAQ**: uses a mixture of several prediction models
  - PPM-like  $n$ -gram contexts
    - Various sparse and bit-twiddled subsets of such contexts
  - Periodic contexts (period is heuristically estimated)
  - Customized models for specific file formats
    - .BMP, .TIFF, .JPEG, .EXE, etc.

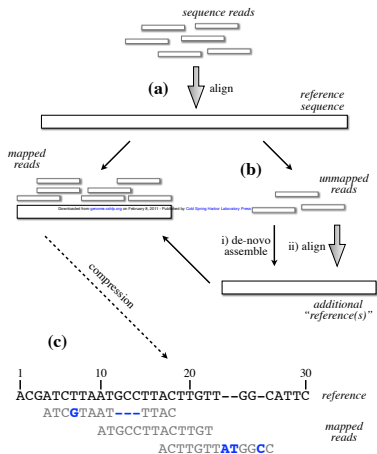
# Transformations of the input string

- Idea: invertible operation, makes input easier to compress
  - e.g. aggregates disperse repetitions
- **Burrows-Wheeler transform**, aka block-sorting
  - Sort all rotations of text, then take last column
    - Invertible permutation of input string
  - Also has some excellent properties as a **substring index**
    - **FM Index** — **Opportunistic Data Structures with Applications** Ferragina and Manzini (2000)
- Turns repeated motifs  $\leftrightarrow$  repeated characters
  - Can then compress e.g. with run-length encoding or order- $N$  Markov

## Motivation: whole genome re-sequencing

- Example: **cancer re-sequencing** (**International network of cancer genome projects**, ICGC, Nature 2010)
  - At least 10 cancer resequencing projects by 2010 (many more coming)
  - 300-500 samples per tumour
  - Sequencing coverage at least 30X
  - That's about  $5 \times 10^{13}$  bases, per project
  - FASTQ uses 8 bits for the base + 8 bits for the **quality score**
  - So,  $10^{14}$  bytes  $\simeq$  60 terabytes per project. OUCH
- Could throw the data away after analysis.... unsatisfactory
- **Most of the reads are substrings of the reference genome**

# Compression of reads to reference genome



Birney *et al.* 2011.



# Compression of reads to reference genome

- Central idea:
  - **Align reads to reference genome.**
    - For unaligned reads, do *de novo assembly*, then align to that
  - Store read lengths using **Huffman coding**.
  - Store distances between reads/differences using **Golomb coding** ( $m$  = expected distance between reads/differences)
  - Read-pairs: use Golomb for separation, plus 3 extra bits to indicate strands & relative orientation of paired reads.
  - **Lossy compression** of quality scores.
- References:
  - Birney *et al.* **Efficient storage of high throughput sequencing data using reference-based compression.** Genome Research, 2011.
  - Baldi *et al.* **Data structures and compression algorithms for high-throughput sequencing technologies.** BMC Bioinformatics, 2010.

## Underlying structure of Baldi/Birney methods

- **Augment** data before **compressing** it.
  - Augmented data is easier to compress
    - e.g. as a delta from a known reference genome
  - Augmentation step involves (CPU-intensive) processing
  - In this case augmentation = alignment (+ assembly)
  - Could also imagine **phylogenetic placement** as augmentation step
- Note that augmentation is theoretically unnecessary—we could marginalize the augmentation and shave off some bits—but this may be impractical, or even impossible due to complexity

## Some other compression methods for genomes

- NB these do not involve augmentation *or* short reads. Just “straightforward” compression of genomes
  - Matsumoto *et al*, 2000. **Biological sequence compression algorithms.** Finds repeats, palindromes; c.f. LZ77, LZ78
  - Chen *et al*, 2002. **DNACompress: fast and effective DNA sequence compression.** Finds repeats
  - Christley *et al*, 2009. **Human genomes as email attachments.** James Watson’s genome in 4MB

# The Right Way To Do It

- Proposal: **bio-oriented compression library**
  - Arithmetic coding/decoding
  - Handlers for standard file formats
  - Preprocessing: assembly, alignment, annotation, phylogeny
  - Models: phylogenetic factor graphs, HMMs, SCFGs
  - Modular reference dictionaries that are themselves compressed

# Summary

- Compressed bits/symbol  $\geq$  Shannon entropy/symbol
  - Arithmetic coding approaches limit, given adaptive model:

$$P(\text{next symbol}|\text{previous symbols})$$

- Other codes (e.g. Golomb, Huffman, Lempel-Ziv) are easier to implement & often run faster
  - All codes have an implicit probability distribution for which they are ideal
- Transformation and preprocessing often useful
  - e.g. aligning high-throughput sequence reads to reference genome