

Topic 3: Lossless Coding

Can view as a special case of general lossy coding problem

Optimal 0 distortion coding.

$\{X_n\}$ stationary random process

Alphabet $A = \{a_0, \dots, a_{M-1}\}$,

marginal pmf $p(a) = p_X(a) = \Pr(X_n = a)$.

E.g., $X_n = Q(W_n)$ are quantized versions of a continuous alphabet sequence W_n .

Alphabet can itself be a vector space.

Need perfectly lossless, invertible.

Simple binary invertible codes

Consider fixed rate or fixed length code:

Input Letter	Codeword
a_0	00
a_1	01
a_2	10
a_3	11

Can uniquely decode individual codewords and sequences if know where blocks start. E.g., 0000011011 decodes as $a_0a_0a_1a_2a_3$.

A real world example:

ASCII (American Standard Code for Information Interchange): assigns binary numbers to all letters, numbers, punctuation for use in computers and digital communication and storage.

All letters have same number of binary symbols (7).

Example:

a \rightarrow 110001

b \rightarrow 110010

c \rightarrow 110011

d \rightarrow 110100

e \rightarrow 110101

\vdots

May get compression.

E.g., if represent each letter by one byte (8 bits), code to only 7.

But clearly need at least R bits, where $2^R > M$.

Generally a fixed rate code gives no lossless compression unless original representation was inefficient. To get lossless compression need a variable length code.

variable rate
ambiguous

Input Letter	Codeword
a_0	0
a_1	10
a_2	101
a_3	0101

Code cannot always be decoded in a noiseless fashion when the code is applied to a sequence of inputs.

E.g., 0101101010... could be produced by $a_0a_2a_2a_0a_1 \dots$ or by $a_3a_2a_0a_1 \dots$.

Ambiguity can never be resolved regardless of future received bits.

Could render unambiguous with addition of third symbol, e.g., the space between the letters in Morse code.

Morse Code “compresses” by using fewer binary numbers for frequently occurring letters (e , t), more for rare letters (z , q)

$e \rightarrow \text{dot } or \cdot or 0$

$t \rightarrow \text{dash } or - or 1$

$c \rightarrow \text{dash dot dash dot } or - \cdot - \cdot or 1010$

Compression results since *on the average* only need about $1/2$ the binary symbols on the average.

But this wasteful, does not make efficient use of three symbol alphabet if one symbol used only for punctuation.

Better to require code be *uniquely decodable*: if decoder receives a valid encoded sequence of finite length, there is only one possible input sequence.

i.e., lossless coding of *sequence*.

variable rate
uniquely decodable

Input Letter	Codeword
a_0	0
a_1	10
a_2	110
a_3	111

If know where start, all valid encoded sequences of codewords have unique decoding.

Lossless Coding Fundamentals

A *noiseless code* or *lossless code* consists of

- an encoder $\alpha : A \rightarrow \mathcal{W} \subset \{0, 1, \}^*$
binary codeword has length $l(\alpha(x))$
and
- a decoder $\beta : \mathcal{W} \rightarrow A$ such that $\beta(\alpha(x)) = x$.

One-to-one, invertible

average length

$$\bar{l}(\alpha) = El(\alpha(X)) = \sum_{a \in A} p(a)l(a). \quad (6)$$

- How small can $\bar{l}(\alpha)$ be?
- How achieve performance close to optimal if
no constraint on complexity or
implementation?
- How achieve near optimal performance in an
implementable way?

Prefix-free and Tree-structured Codes

Prefix condition: No codeword is a prefix of another codeword.

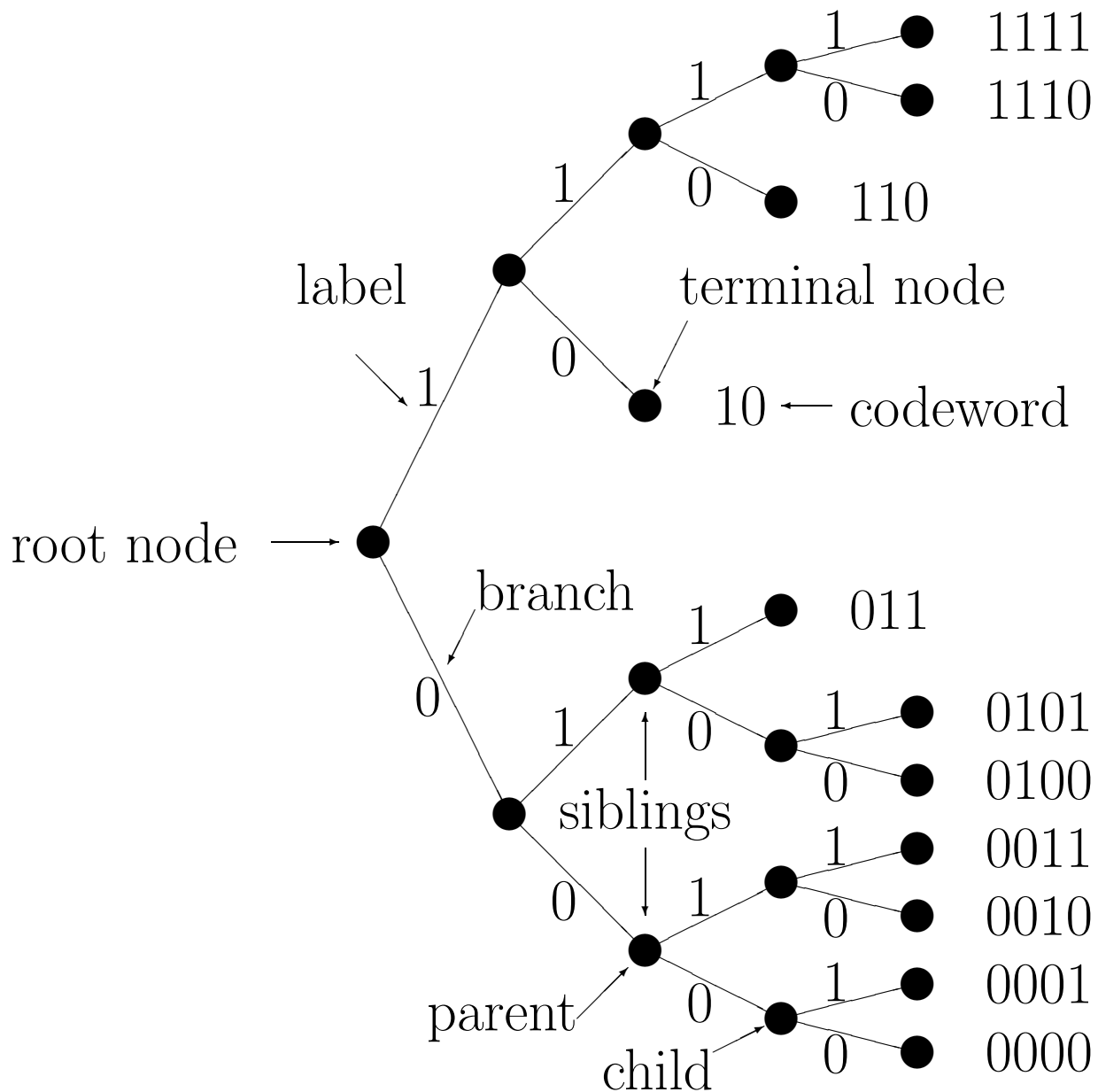
Such a code is said to be *prefix-free*.

Ensures uniquely decodable.

Essentially no loss of generality:

As will see, for any uniquely decodable code there is a tree-structured code with the same collection of codeword lengths and the same average length.

Binary prefix-free codes can be depicted as a binary *tree*:



Theorem 1 (*The Kraft Inequality*)

If a uniquely decodable code has lengths l_k ; $k = 0, 1, \dots, M - 1$, then necessarily these lengths satisfy the Kraft inequality

$$\sum_{k=0}^{M-1} 2^{-l_k} \leq 1. \quad (7)$$

Furthermore, given any collection of lengths l_k ; $k = 0, 1, \dots, M - 1$ which satisfies the Kraft inequality, there exists a uniquely decodable code for an alphabet $\{a_0, \dots, a_{M-1}\}$ having these lengths.

Proof of necessity: In lots of books.

See, e.g., Cover & Thomas, Gersho & Gray

Proof of sufficiency:

Can use tree to construct prefix-free code.

Label so $l_0 \leq l_1 \leq \dots \leq l_{M-1}$

Draw full binary tree.

Pick arbitrary length l_0 binary codeword in tree.

(0, length 1 in figure)

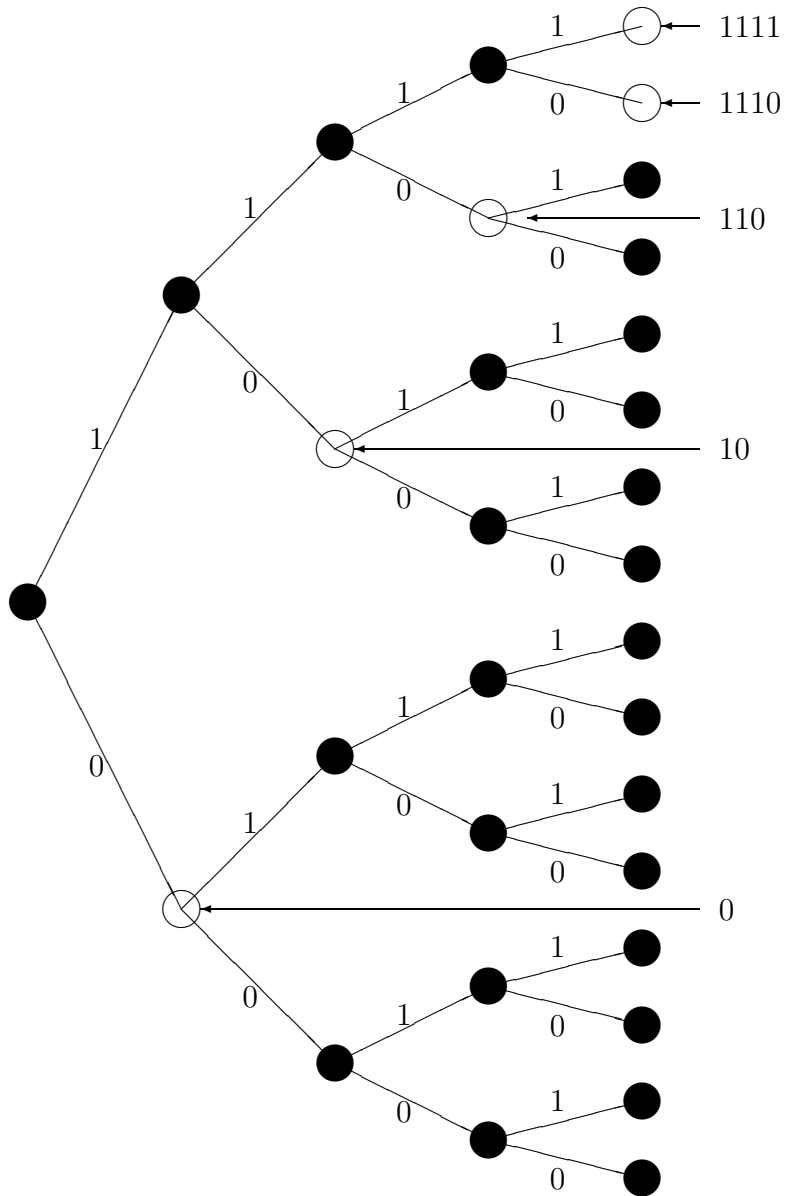
Prune the tree at terminating node (remove all branches extending from the node).

This removes $2^{l_{M-1}-l_0}$ leaves of original tree.

Next pick arbitrary length l_1 binary codesword

(10, length 2)

Remove $\sum_{k=0}^{M-1} 2^{l_{M-1}-l_k}$ leaves during pruning, less than $2^{l_{M-1}}$ available by Kraft.



Examples: of testing Kraft inequality:

- 1,2,3,3

$$\sum_i 2^{-l_i} = 1$$

- 2,2,2,2

$$\sum_i 2^{-l_i} = 1$$

- 1,1,2,2

$$\sum_i 2^{-l_i} = 1.5$$

Since any uniquely decodable code has lengths which satisfy the Kraft inequality, and for any collection of lengths satisfying the Kraft inequality we can construct a prefix code as in the sufficiency proof, given any UDC we can find a prefix-free (tree-structured) code with the same collection of lengths.

Every prefix-free code is uniquely decodable. Is the converse true?

No: $M=3$, codewords = 0,01,11

Uniquely decodable, but not a prefix code.

Entropy

From Kraft inequality

$$\begin{aligned}\bar{l}(\alpha) &= \sum_{a \in A} p(a) l(a) \\ &= - \sum_{a \in A} p(a) \log_2 2^{-l(a)} \\ &\geq - \sum_{a \in A} p(a) \log_2 \frac{2^{-l(a)}}{\sum_{b \in A} 2^{-l(b)}},\end{aligned}\tag{8}$$

where the logarithm is base 2.

Lower bound has form

$$\sum_{a \in A} p(a) \log \frac{1}{q(a)}$$

for two pmf's p and q .

A famous result shows that this term is further bounded below by the *entropy* of the random variable, defined by

$$H(p) \equiv \sum_{a \in A} p(a) \log \frac{1}{p(a)}$$

This result is easy to prove and we do so via one of the fundamental inequalities of information theory:

Theorem 2 *The Divergence Inequality:*

Given any two pmf's p and q with a common alphabet A , then

$$H(p||q) \equiv \sum_{a \in A} p(a) \log \frac{p(a)}{q(a)} \geq 0 \quad (9)$$

or, equivalently,

$$\sum_{a \in A} p(a) \log \frac{1}{q(a)} \geq H(p) \equiv \sum_{a \in A} p(a) \log \frac{1}{p(a)}$$

Equality holds in (9) if and only if $p(a) = q(a)$ for all $a \in A$.

$H(p||q)$ called the *divergence* or *relative entropy* or *cross entropy* or *Kullback-Leibler number* of the pmf's p and q .

$H(p)$ is called the *entropy* of the pmf p or, equivalently, the entropy of the random variable X described by the pmf p . Both notations $H(p)$ and $H(X)$ are common.

Proof: Change the base of logarithms and use the elementary inequality $\ln r \leq r - 1$ (with equality if and only if $r = 1$):

$$\begin{aligned}
 \sum_{a \in A} p(a) \log \frac{q(a)}{p(a)} &= \frac{1}{\ln 2} \sum_{a \in A} p(a) \ln \frac{q(a)}{p(a)} \\
 &\leq \frac{1}{\ln 2} \sum_{a \in A} p(a) \left(\frac{q(a)}{p(a)} - 1 \right) \\
 &= \frac{1}{\ln 2} \left(\sum_{a \in A} q(a) - \sum_{a \in A} p(a) \right) \\
 &= 0.
 \end{aligned}$$

The inequality is an equality if and only if $q(a) = p(a)$ for all a , which completes the proof. \square

Alternative proof:

Jensen's inequality states that if $f(x)$ is convex \cap , i.e., $f(\lambda x + (1 - \lambda)y) \geq \lambda f(x) + (1 - \lambda)f(y)$ for all $0 < \lambda < 1$, then $E(f(X)) \leq f(E(X))$. (With equality iff X is constant with probability 1.)

$\ln x$ is a convex \cap function. So (expectation with respect to p :

$$E(\ln \frac{q(X)}{p(X)}) \leq \ln E(\frac{q(X)}{p(X)}) = \ln(\sum_x p(x) \frac{q(x)}{p(x)}) = 0.$$

□

Corollary 1 *If r.v. X has K possible symbols, then*

$$H(X) \leq \log K.$$

Proof: Let $q(a)$ be uniform distribution and follows from divergence inequality.

Combining Theorem 2 with (8) yields

Theorem 3 *Given a uniquely decodable scalar noiseless variable length code with encoder α operating on a random variable X with entropy $H(X)$, then the resulting average codeword length satisfies*

$$E[l(\alpha(X))] \geq H(X); \quad (10)$$

that is, the average length of the code can be no smaller than the entropy of the marginal pmf. The inequality is an equality if and only if

$$p(a) = 2^{-l(a)} \text{ for all } a \in A. \quad (11)$$

The equality (11) follows when both (9) and (8) hold with equality. The latter equality implies that $q(b) = 2^{-l(b)}$.

To achieve the lower bound, need to have (11) satisfied. Can only hold if all probabilities that are powers of $1/2$.

Considered “ideal” codeword lengths, but not always possible.

We can never code to average length lower than the entropy, but can we actually achieve the entropy?

Theorem 4 *There exists a uniquely decodable scalar noiseless code for a source with marginal pmf p for which the average codeword length satisfies*

$$\bar{l}(\alpha) < H(p) + 1. \quad (12)$$

Proof: Source probabilities:

$$p_0, p_1, \dots, p_{M-1}.$$

For $k = 0, 1, \dots, M - 1$ choose l_k as the integer satisfying

$$2^{-l_k} \leq p_k < 2^{-l_k+1} \quad (13)$$

(sandwich p_k) or, equivalently,

$$-\log p_k \leq l_k < -\log p_k + 1. \quad (14)$$

l_k is the smallest integer $\geq -\log p_k$.

Note that this can be interpreted as a rough approximation to the ideal codeword lengths:

$$l_k \approx -\log p_k$$

Lengths must satisfy the Kraft inequality since

$$\sum_{k=0}^{M-1} 2^{-l_k} \leq \sum_{k=0}^{M-1} p_k = 1,$$

hence there is a prefix-free (and hence a uniquely decodable) code with these lengths. From (14), the average length must satisfy the bound of the theorem. \square

Combining these two results yields Shannon's lossless coding theorem:

For any uniquely decodable lossless code

$$\bar{l}(\alpha) \geq H(p); \quad (15)$$

and there exists a prefix-free code for which

$$\bar{l}(\alpha) < H(p) + 1. \quad (16)$$

Prefix-free Codes

Previous results imply several properties that will yield a constructive coding technique.

Theorem 5 *Suppose that (α, β) is a uniquely decodable variable length noiseless source code and that*

$\{l_m; m = 0, 1, \dots, M - 1\} = \{l(a); a \in A\}$ is the collection of codeword lengths. Then there is a prefix-free code with the same lengths and the same average length.

Follows since the lengths of a uniquely decodable code must satisfy the Kraft inequality and hence there must exist a prefix-free code with these lengths (from tree construction of sufficiency proof).

The theorem implies that the optimal prefix-free code is as good as the optimal uniquely decodable code.

Theorem 6 *An optimum binary prefix-free code has the following properties:*

- (i) If the codeword for input symbol a has length $l(a)$, then $p(a) > p(b)$ implies that $l(a) \leq l(b)$; that is, more probable input symbols have shorter (at least, not longer) codewords.*
- (ii) The two least probable input symbols have codewords which are equal in length and differ only in the final symbol.*

Proof:

- (i) If $p(a) > p(b)$ and $l(a) > l(b)$, then exchanging codewords will cause a strict decrease in the average length. Hence the original code could not have been optimum.*

(ii) Suppose that the two codewords have different lengths. A prefix of longer codeword cannot itself be a codeword, can delete the final symbol of the longer codeword without confusion. This strictly decreases the average length of the code \neq . Thus the two least probable codewords must have equal length.

Suppose that these two codewords differ in some position other than the final one. Could then remove the final binary symbol and shorten the code without confusion. (could still distinguish the shorter codewords and prefix condition precludes the possibility of confusion with another codeword). This, however, would yield a strict decrease in average length \neq . □

The theorem provides an iterative design technique for optimal codes, developed by D. A. Huffman (1952)

Note “optimum” for specified code structure, i.e., memoryless operation on each successive symbol.

Huffman Coding

D. A. Huffman (1952) developed a scheme which yields performance quite close to the lower bound of Theorem 1. If the input probabilities are powers of $1/2$, the bound is achieved.

Order the input symbols in terms of probability, that is, $p(a_0) \geq p(a_1) \geq \cdots \geq p(a_{M-1})$.

The two least probable symbols have codewords of the same length which differ only in the final binary symbol.

Begin a code tree with two terminal nodes with branches extending back to a common node.

Label one branch 0 and the other 1. We now consider these two input symbols to be *tied* together and form a single new symbol in a reduced alphabet A' with $M - 1$ symbols in it.

Next find an optimal code for the reduced alphabet A' with probabilities

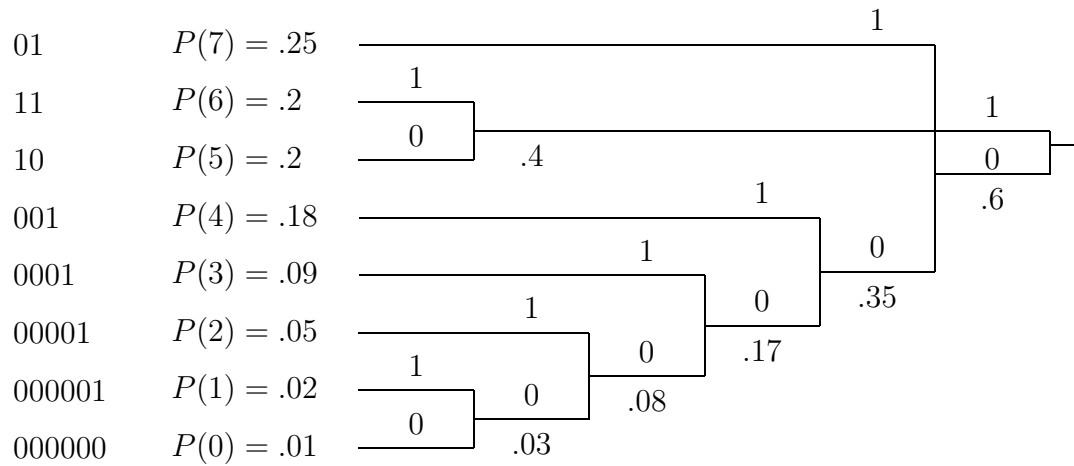
$p(a_m)$; $m = 0, 1, \dots, M - 3$ and $p(a_{M-1}) + p(a_{M-2})$.

A prefix code for A' implies a prefix code for A by adjoining the final branch labels already selected. If the prefix code for A' is optimal, then so is the induced code for A .

Continue in this fashion:

- probability of each node is found by adding up the probabilities of all input symbols connected to the node.
- At each step the two least probable nodes in the tree are found.
- These nodes are tied together and a new node is added with branches to each of the two low probability nodes and with one branch labeled 0 and the other 1.
- The procedure is continued until only a single node remains (the list contains a single entry).

Example:



LARGE

A Huffman Code

Variation on this technique works for nonbinary alphabets (and can do better).

Vector Entropy Coding

Could code vectors of input symbols instead of single symbols.

Similar bounds for higher order entropy.

Same construction works.

p_{X^N} is the pmf for a source vector

$X^N = (X_0, \dots, X_{N-1})$, then a uniquely decodable noiseless source code for successive source blocks of length N has an average codeword length no smaller than the N th order entropy $H(X^N)$ of the input defined by

$$H(X^N) = H(p_{X^N}) = - \sum_{x^N} p_{X^N}(x^N) \log p_{X^N}(x^N).$$

Shannon:

For any integer N average length satisfies

$$\bar{l} \geq \frac{1}{N} H(X^N). \quad (17)$$

There exists a prefix-free code for which

$$\bar{l} < \frac{1}{N} H(X^N) + \frac{1}{N}. \quad (18)$$

For any fixed N , Huffman optimal.

It can be shown that if the input process is stationary, then minimum over N on the right hand side achieves a lower bound for all N and this minimum is \overline{H} , the *entropy rate* of the source

$$\overline{H} = \lim_{N \rightarrow \infty} \frac{H(X^N)}{N}.$$

If the source is iid, then $\overline{H} = H(X_0)$.

If “optimal” and can get arbitrarily close to theoretical limit, why not always use?

Because too complicated: Big N needed, which can make Huffman tables enormous.
(Exponential growth with blocksize.)

Also: Do not always know distributions, and they can change.

Leads to many methods that are “suboptimal,” but which outperform any (implementable) Huffman code.

Good ones are also asymptotically optimal, e.g., arithmetic and Lempel-Ziv.

Note: codes have global block structure on a sequence, but locally they operate very differently. Nonblock, memory

Two general approaches:

- Statistical
Huffman, arithmetic
- Dictionary
Lempel-Ziv

Consider each.

Arithmetic codes:

Finite-precision version of Elias code developed by Pasco and Rissanen.

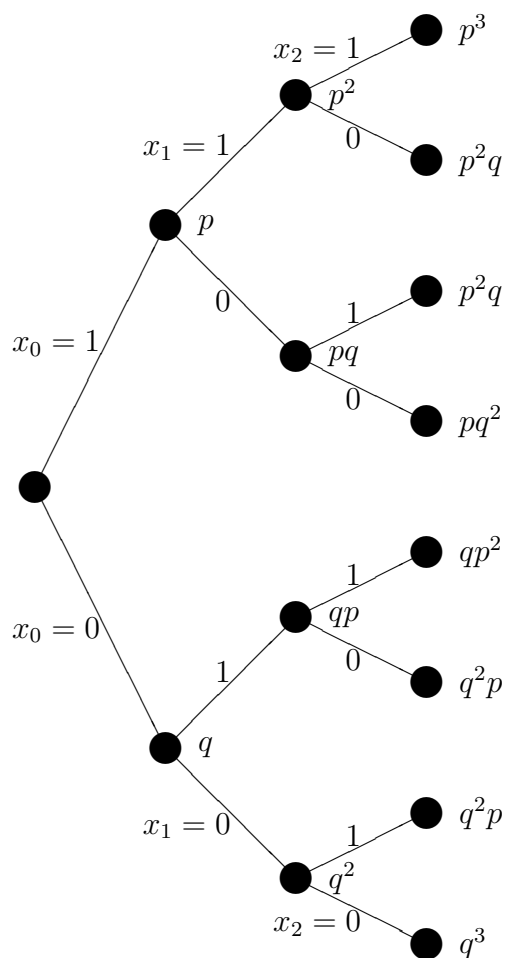
For details see Chapter 5 of *Text Compression*, by Bell, Cleary, and Witten or Chapter 4 of *Introduction to Data Compression*, by Sayood. As in Gersho & Gray Chapter 9, here focus on original Elias code.

Arithmetic code is a finite precision approximation to an Elias code.

Here consider only special case of lossless coding a binary iid source $\{X_n\}$ with $\Pr(X_n = 1) = p$ and $\Pr(X_n = 0) = q$.

(It is convenient to give a separate name to $q = 1 - p$.)

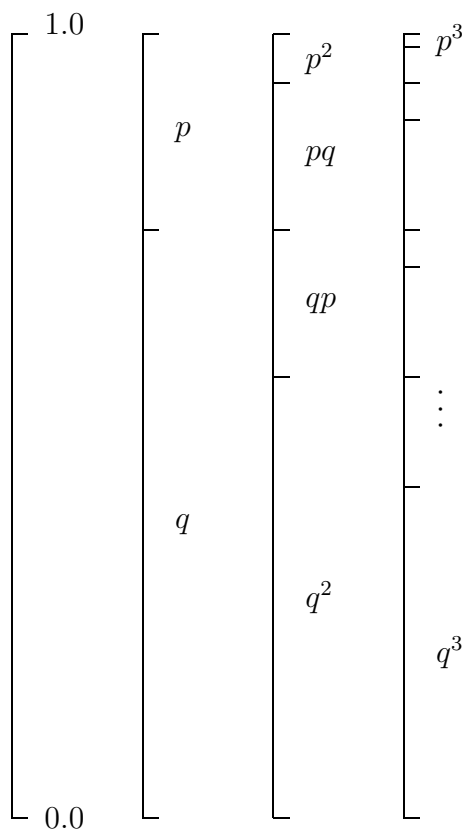
Key idea: Produce a code that results in codelengths near the Shannon optimal, i.e., code a sequence $x^n = (x_0, x_1, \dots, x_{n-1})$ into a codeword u^L where $L = l(x^n) \approx -\log p_{X^n}(x^n)$.

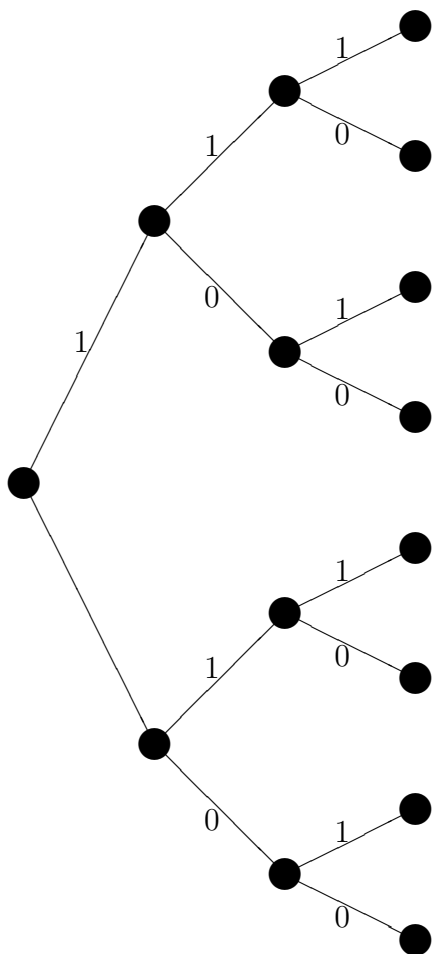


SOURCE TREE

x^n corresponds to
interval $I_n(x^n) = I_n = [a_n, b_n)$
such that $|I_n| = b_n - a_n = p_{X^n}(x^n)$

$$[a_n, b_n) = \begin{cases} [a_{n-1}, a_{n-1} + q(b_n - a_n)) & \text{if } x_n = 0 \\ [a_{n-1} + q(b_n - a_n), b_{n-1}) & \text{if } x_n = 1 \end{cases}$$

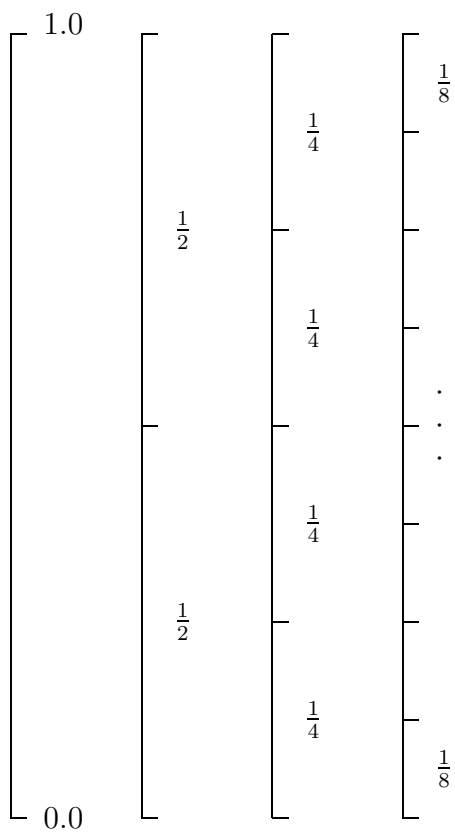




CODE TREE

u^L corresponds to
interval J_L such that $|J_L| = 2^{-L}$

$$J_L = [\sum_{l=0}^{L-1} u_l 2^{-l-1}, \sum_{l=0}^{L-1} u_l 2^{-l-1} + 2^{-L})$$



Suppose encoder and decoder both know n

Encoder: $x^n \rightarrow I_n \rightarrow J_L \subseteq I_n \rightarrow u^L$

Lots of ways to choose L and J_L . Choose smallest possible L and hence biggest possible interval J_L .

$$|I_n| \geq |J_L| = 2^{-L} \geq \left| \frac{I_n}{2} \right| \quad (\star)$$

Reason: Given I_n , divide unit interval into 2^L subintervals of size 2^{-L} where L chosen as smallest integer for which $|I_n| \geq 2^{-L}$. Then one of the subintervals must be contained in I_n and hence $|J_L| \geq \left| \frac{I_n}{2} \right|$

Decode: $u^L \rightarrow J_L \rightarrow I_n \supseteq J_L \rightarrow x^n$

Key fact: From (\star) we have taking logarithms that

$$-\log |I_n| \leq L \leq -\log \left| \frac{I_n}{2} \right|$$

$$-\log p_{X^n}(x^n) \leq L \leq -\log \frac{p_{X^n}(x^n)}{2} = -\log p_{X^n}(x^n) + 1$$

Thus if $x^n \rightarrow J_L$, then from (\star)

$$-\log p_{X^n}(x^n) \leq l(\alpha(x^n)) \leq -\log p_{X^n}(x^n) + 1$$

so $l(x^n) \approx -\log p_{X^n}(x^n)$ as desired.

Taking expectations and normalizing:

$$\frac{1}{n}H(X^n) \leq El(X^n) \leq \frac{1}{n}H(X^n) + \frac{1}{n}$$

which converges to the optimum \overline{H} as n gets large! (but have proved only for iid)

Collection of leaves $u^L(x^n)$ is a prefix-free code since a subtree.

Why “better” than Huffman?

Incrementality of algorithm permits much larger n than does Huffman. E.g., suppose $n = 256 \times 256$ for a modest size image.

Huffman out of the question, would need a tree of depth 2^{65536} .

Elias encoder can compute $[a_k, b_k)$ incrementally as $k = 1, 2, \dots, n$.

As I_k shrinks, can send “release”, “committ” bits incrementally: Send up to l bits where

$$J_l \supseteq I_k,$$

J_l is the smallest interval containing I_k .

Decoder receives J_l , can decode up to x^m , where

$$I_m \supseteq J_l \supseteq I_k$$

where $m < k$.

not instantaneous code, variable number of bits needed to decode symbols.

Finally, encoder sends final bits, or can stop if no ambiguity at decoder (knowing n).

- practical implementations need finite precision, result is arithmetic code.
- trees can be m -ary

Extend to sources with memory by using
“context modeling”

Compute

$$[a_n, b_n) =$$

$$[a_{n-1}, a_{n-1} + (b_{n-1} - a_{n-1}) \Pr(X_n = 0|x^{n-1}))$$

if $x_n = 0$, or

$$[a_{n-1} + (b_{n-1} - a_{n-1}) \Pr(X_n = 0|x^{n-1}), b_{n-1})$$

if $x_n = 1$

$\Pr(X_n = 0|x^{n-1}))$ can be learned on the fly, e.g.,
assume Markov order and estimate from
histograms.

Universal and Adaptive Entropy Coding:

Both Huffman codes and arithmetic codes assume *a priori* knowledge of the input probabilities. Often not known in practice.

- Universal Codes: Collection of codes, pick best.
- Adaptive Codes: Adjust code parameters in real time. Often by fitting model to source and then coding for model.

Example of Universal Codes: Rice machine
(multiple codebooks, pick one that does best
over some time period).

Lynch-Davisson codes, Cover's enumerative
codes: View binary N -vector. First send integer
 n giving the number of ones (this is an estimate
 n/N of underlying p), then send binary vector
index saying which of the $\binom{N}{n}$ weight n
 N -dimensional vectors was seen.

Lempel-Ziv Coding

Inherently universal.

Variable numbers of input symbols are required to produce each code symbol.

Unlike both Huffman and arithmetic codes, the code does not use any knowledge of the probability distribution of the input. As with the Elias code, the Ziv-Lempel code achieves the entropy lower bound when applied to a suitably well-behaved source.

Basic idea: recursively parse input sequence into nonoverlapping blocks of variable size while constructing a dictionary of blocks seen thus far. Each sequence found in the dictionary is coded into its index in the dictionary.

The dictionary is initialized with the available single symbols, here 0 and 1.

Each successive block in the parsing is chosen to be the largest (longest) word ω that has appeared in the dictionary and hence the word ωa formed by concatenating ω and the following symbol is not in the dictionary.

Before continuing the parsing ωa is added to the dictionary and a becomes the first symbol in the next block.

Example of parsing and dictionary construction:

01100110010110000100110.

parsed blocks enclosed in parentheses and the new dictionary word (the parsed block followed by the first symbol of the next block) written as a subscript:

$(0)_{01}(1)_{11}(1)_{10}(0)_{00}(01)_{011}(10)_{100}(01)_{010}(011)_{0110}$
 $(00)_{000}(00)_{001}(100)_{1001}(11)_{110}(0).$

The parsed data implies a code by indexing the dictionary words in the order in which they were added and sending the indices to the decoder.

Details for input sequence

01100110010110000100110

Input String	Index
0	0
1	1

Initial Code Table

Input String	Index
0	0
1	1
01	2

Step 1: Longest string in table: 0, Output: 0,
Add to table: 01, pointer: 2

Input String	Index
0	0
1	1
01	2
11	3

Step 2: Longest string in table: 1, Output: 1,
Add to table: 11, pointer: 3

Input String	Index
0	0
1	1
01	2
11	3
10	4

Step 3: Longest string in table: 1, Output: 1,
Add to table: 10, pointer: 4

Input String	Index
0	0
1	1
01	2
11	3
10	4
00	5

Step 4: Longest string in table: 0, Output: 0,
Add to table: 00, pointer: 5

Input String	Index
0	0
1	1
01	2
11	3
10	4
00	5
011	6

Step 5: Longest string in table: 01, Output: 2,
Add to table: 011, pointer: 7

Input String	Index
0	0
1	1
01	2
11	3
10	4
00	5
011	6
100	7

Step 6: Longest string in table: 10, Output: 4,
Add to table: 100, pointer: 9

Input String	Index
0	0
1	1
01	2
11	3
10	4
00	5
011	6
100	7
010	8

Step 7: Longest string in table: 01, Output: 2,
Add to table: 010, pointer: 11

Things start slowly as the table builds.

Observe that the input words in the table are always prefixes of longer words as the longer words are added.

At the completion of all the steps in the tables, the encoder has produced the sequence of integers (or N -dimensional binary vectors)

0110242.

Before describing the operation of the decoder we observe an important property of the encoder:

The Last-First Property: The *last* symbol of the most recent word added to the table is the *first* symbol of the next parsed vector.

Decoder starts with the same initial table and hence immediately decodes the first symbol as 0 and the second as 1.

The decoder recognizes the new string 01 and adds it to its table with index 2.

The next symbol is a 1, \rightarrow an input of 1 was seen by the encoder. The decoder sees the string 11 terminating with the current symbol and adds it to its table with index 3.

On seeing the fourth symbol 0, the decoder again knows the encoder saw a 0 and it recognizes the sequence 10 terminating with the current symbol as a new one for its table with index 4.

On encountering the code symbol 2, the decoder knows that the encoder saw the pattern 01 and decodes this pattern. It also recognizes the previously decoded 0 and the first 0 of this new decoded pattern as a 00, which must be added to the table with index 5.

The next symbol is a 4, which means that the decoder produces a 10. The previous pattern 01 combined with the first symbol of the new pattern produces a 011, which is added to the table with index 6.

In this way the decoder builds its copy of the encoder table. The question marks denote symbols that cannot yet be determined.

In	Out	Reconstructed Sequence	Add to Table
0	0	(0) _{0,?}	
1	1	(0) _{0,1} (1) _{1,?}	(0,1) (as 2)
1	1	(0) _{0,1} (1) _{1,1} (1) _{1,?}	(1,1) (as 3)
0	0	(0) _{0,1} (1) _{1,1} (1) _{1,0} (0) _{0,?}	(1,0) (as 4)
2	(0,1)	(0) _{0,1} (1) _{1,1} (1) _{1,0} (0) _{0,0} (01) _{0,1,?}	(0,0) (as 5)
4	(1,0)	(0) _{0,1} (1) _{1,1} (1) _{1,0} (0) _{0,0} (01) _{0,1,1} (10) _{1,0,?}	(0,1,1) (as 6)

Decoder

Problem with the algorithm as described. There exists a type of sequence which at first glance appears confusing to the decoder. Consider a ternary source with symbols $\{0, a, b\}$ and an input sequence

$$a000ba0a \cdots$$

The initial table is

Input String	Index
0	0
a	1
b	2

Initial Code Table

The encoder will parse this sequence as

$$(a)_{a,0}(0)_{0,0}(0,0)_{0,0,b}(b)_{b,a}(a0)_{a,0,a} \cdots$$

Time	Send	New Entry	Index
1	1 (for 0)	$(a,0)$	3
2	0 (for 0)	$(0,0)$	4
3	4 (for $(0,0)$)	$(0,0,b)$	5
4	2 (for b)	$(b,0)$	6
5	3 (for $(a,0)$)	$(a,0,a)$	7

Ternary Encoder

Decoder then begins as previously described

In	Out	Reconstructed Sequence	Add to Table
1	a	$(a)_{a,?}$	
0	0	$(a)_{a,0}(0)_{0,?}$	$(a,0)$ (as 3)
4	?	$(a)_{a,0}(0)_{0,?}$?

Ternary Decoder

Problem: The decoder receives an index for table entry 4, but the entry does not yet exist in the table! Without additional guidance, the decoder is stuck.

This behavior can arise whenever one sees a pattern of the form $x\omega x\omega x$, where x is a single symbol and ω is either empty or a sequence of symbols such that $x\omega$ already appears in the encoder and decoder table, but $x\omega x$ does not. In our example $x = 0$ and ω is empty.

The encoder will send the codeword for $x\omega$ and add $x\omega x$ ((0,0) in the example) to the table with a new index i (4 in the example).

Next it will parse $x\omega x$ and send the new index i corresponding to the just added word.

The decoder will receive the index i for $x\omega x$, but will not yet have this word in its table because it has not yet determined the character that terminates the previously received string $x\omega$ to complete the previously added table entry.

Thus the decoder knows that i corresponds to a new table entry of the form $x\omega?$, where $?$ is the input signal that followed the $x\omega$ previously decoded. But the final symbol $?$ of the previously added table entry must be the first symbol in the current block being decoded using the Last-First Property, which the decoder knows is an x . This tells the decoder that the new encoder entry is $x\omega x$ and the decoder can proceed.

In the example, the decoder knows that entry 4 must be an extension of the previously decoded string, that is, have the form $x?$ where x is the last encoded string, 0 in the current example, and $?$ is a single letter. The Last-First Property implies that the last letter ($?$ here) of the last entry must be the first letter of the current decoded string, $x = 0$ in the current example. Thus 4 must be $(0, 0)$ and decoding can proceed. This fix is due to Welch and the code sometimes called the LZW code.

The code is noiseless and can be shown to be optimum in the limit of unbounded size. The disadvantage of the algorithm is that unmanageably large tables may be required in some applications in order to achieve the desired performance. (Can use static or dynamically adapt.)