Data Compression

Chapter 5, Elements of Information Theory

(Binary) Source Codes

We want to efficiently encode (in bits) sequences produced by a random source variable $X \sim p(x)$.

The expected length of a code C is given by:
$L(C) = \sum_x p(x) \cdot \ell(x)$, with $\ell(x)$ the length of $C(x)$.

Example: For $\mathcal{X}=\{a,b,c\}$ we can use the binary source code $C(a)=00$, $C(b)=01$, $C(c)=10$, with $L(C)=2$ bits/letter.

A better idea is to use $C(a)=0$, $C(b)=10$, $C(c)=11$, such that if $p(a) = p(b) = p(c) = 1/3$, the code has expected length 1.66 bits per letter ("bpc").

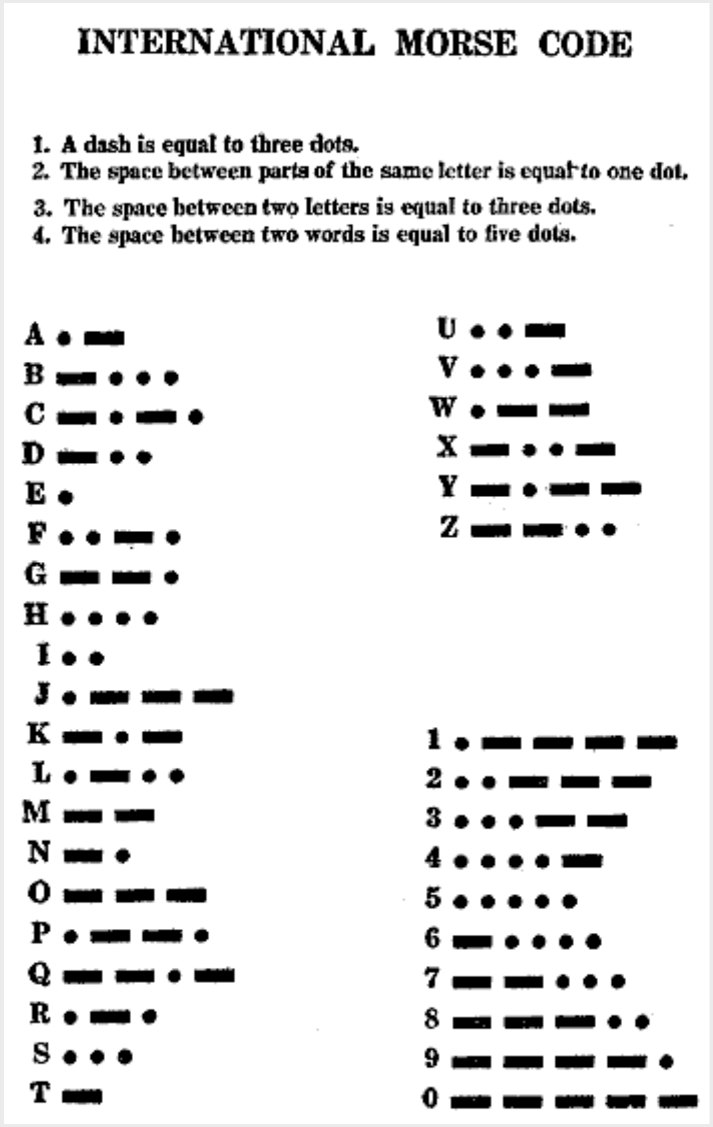
Relying on Bias

Continued example for $\mathcal{X}=\{a,b,c\}$: The C(a)=0, C(b)=10, C(c)=11, works even better if p(a) = ½ and p(b) = p(c) = ¼, in which case the expected length is 1.5 bits per letter.

It is worse if we have p(a) = p(b) = ¼ and p(c) = ½ such that L(C)= ¼ + ½ + 1 = 1.75 bits per letter.

Central idea of variable length coding:
Assign short code words to likely letters
(and longer code words to the less likely letters)

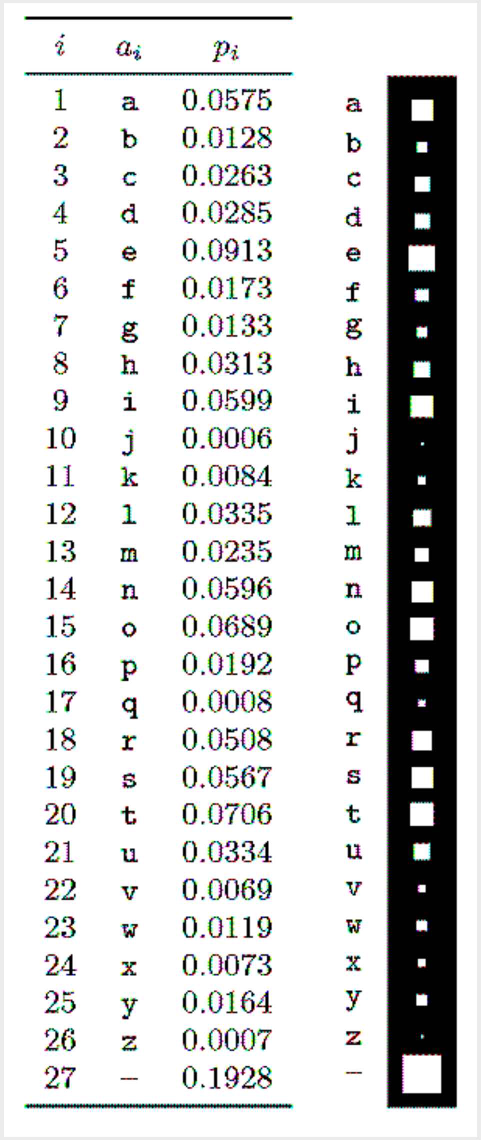
Morse Code



INTERNATIONAL MORSE CODE

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to five dots.

A • —	U • • —
B — • • •	V • • • —
C — • — •	W • — —
D — • •	X — • • —
E •	Y — • — —
F • • — •	Z — — • •
G — — •	
H • • • •	
I • •	
J • — — —	
K — • —	1 • — — — —
L • — • •	2 • • — — —
M — —	3 • • • — —
N — •	4 • • • • —
O — — —	5 • • • • •
P • — — •	6 — • • • •
Q — — • —	7 — — • • •
R • — •	8 — — — • •
S • • •	9 — — — — •
T —	0 — — — — —



i	a_i	p_i
1	a	0.0575
2	b	0.0128
3	c	0.0263
4	d	0.0285
5	e	0.0913
6	f	0.0173
7	g	0.0133
8	h	0.0313
9	i	0.0599
10	j	0.0006
11	k	0.0084
12	l	0.0335
13	m	0.0235
14	n	0.0596
15	o	0.0689
16	p	0.0192
17	q	0.0008
18	r	0.0508
19	s	0.0567
20	t	0.0706
21	u	0.0334
22	v	0.0069
23	w	0.0119
24	x	0.0073
25	y	0.0164
26	z	0.0007
27	–	0.1928

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
–

Distributing Lengths

The question has become: Given a probability p distribution over the alphabet $\mathcal{X}$, how can we assign the lengths $\ell(x)$ of the code words C(x)?

Is it possible to have lengths $\ell_1=1$, $\ell_2=2$, $\ell_3=3$, $\ell_4=3$?

Answer: Yes; use words 0,10,110 and 111.

Is it possible to have lengths $\ell_1=1$, $\ell_2=2$, $\ell_3=2$, $\ell_4=3$?

Answer: No.

Try 0,01,10 and 111; what does 010 mean?

We want our codes to be uniquely decodable.

A Uniquely Decodable Code

Take the following code for $\mathcal{X}$={a,b,c,d}:

but annoying

C(a)=10, C(b)=00, C(c)=11, C(d)=110.

Try decoding the string 100011101100110.

It gives C^{-1}(100011101100110) = abcacbd,
but there was some suspense with the second "c",
as the "11" could also have been the start of C(d)=110.

This code is uniquely decodable,
but the decoding is not instantaneous.

Instantaneous Codes

Definition [EoIT, Section 5.1]:
A code is called a *prefix code* or an *instantaneous code* if no code word is the prefix of any other code word.

We want to do our coding with such prefix codes.

The non-instantaneous code had code lengths 2,2,2,3.

There is an instantaneous code with such lengths.
Take the words 00,01,10,111.

But there is no instantaneous code with lengths 1,2,2,3.

Kraft's Inequality

Crucial question: For which code word lengths $\ell_1,\ldots,\ell_m$ does there exist a prefix code?

Kraft's Inequality: For any binary prefix code with code word lengths $\ell_1,\ldots,\ell_m$, it must hold that:

$$\sum_{j\in\{1,\ldots,m\}} 2^{-\ell_j} \leqslant 1$$

Conversely, if the lengths ℓ_j obey Kraft's inequality, then there is a prefix code with such lengths.

If instead of bits we use the more general Dits $\in \{1,\ldots,D\}$, then we have $\displaystyle\sum_{j\in\{1,\ldots,m\}} D^{-\ell_j} \leqslant 1$

Proving Kraft's Inequality

L.G. Kraft, A device for quantizing, grouping, and coding amplitude modulated pulses, M.Sc. thesis, MIT, 1949.

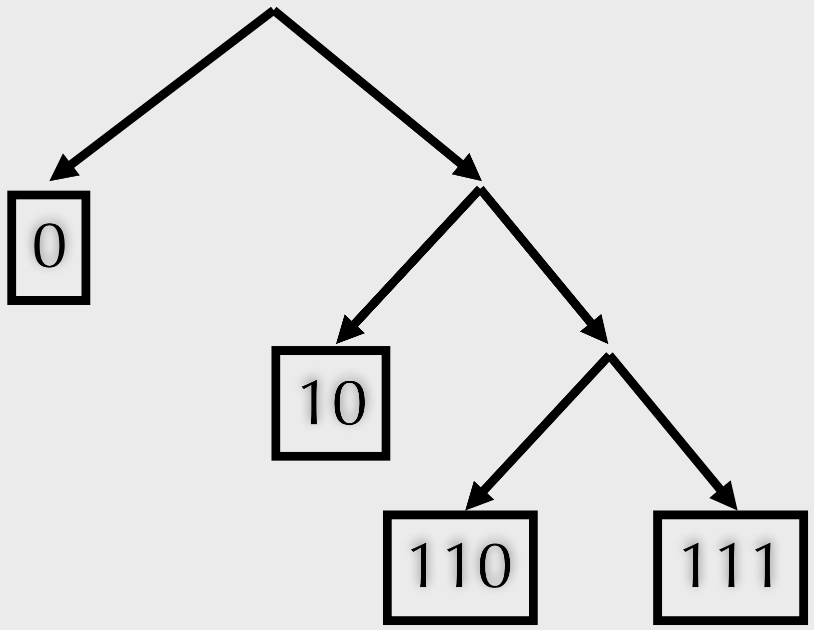
[Cover & Thomas]: "The Kraft inequality… was first proved by McMillan (1956)…"

How to prove it?

It helps to think about prefix codes as trees.

Trees of Prefix Codes

The prefix code $C(\mathcal{X}) = \{0,10,110,111\}$ has as its tree:



Proof of Kraft's Inequality?
See EoIT, Theorem 5.2.1.

Source Coding Question

Given a probability distribution $p_1,\ldots,p_m$, what lengths $\ell_1\ldots,\ell_m$ do we take such that we minimize the expected length $L(C) = \sum_j p_j \cdot \ell_j$, while obeying Kraft's inequality?

Rough answer: Take $\ell_j \approx -\log p_j$ ("amount of surprise")

Note that—if exact—this gives:
$L(C) = \sum_j p_j \cdot \ell_j = \sum_j -p_j \cdot \log p_j = H(p)$.

Using inequalities like $D(p\|q) \geq 0$ we will show that:
- For all prefix codes we must have $L(C) \geq H(p)$.
- There exists a code such that $L(C) < H(p)+1$.

Lossless Coding Theorem

Shannon's 1948 Lossless Source Coding Theorem:
Let X be a random variable with entropy H(X).

For all uniquely decodable codes C we have $L(C) \geq H(X)$.
There exists a prefix code C such that $L(C) < H(X)+1$.

Idea: Let $\mathcal{X}=\{1,\ldots,m\}$. Any code C with lengths $\ell_1,\ldots,\ell_m$ that obeys Kraft's inequality defines a probability distribution q over $\{0,1,\ldots,m\}$ with $q(j) = 2^{-\ell_j}$ (and q(0) such that $\sum_j q(j)=1$).

The expected length of this code: $L(C) = -E_p[\log q(X)]$.

Lower Bound Proof

Let p be the probability distribution of variable X. Let code C have lengths $\ell_1,\ldots,\ell_m$, defining the distribution q over $\{0,1,\ldots,m\}$ with $q(j) = 2^{-\ell_j}$, such that: $L(C) = E_p[-\log q(X)]$. This gives:

$$\begin{aligned} L(C) - H(X) &= E_p[-\log q(X)] - E_p[-\log p(X)] \\ &= E_p[\log p(X)/q(X)] \\ &= D(p\|q) \\ &\geq 0. \end{aligned}$$

The equality $L(C)-H(X) = D(p\|q)$ also shows that we should try to have q=p to get the smallest possible L(C).

What We Know

Kraft's Inequality: For any binary prefix code with code word lengths $\ell_1,\ldots,\ell_m$, it holds that:

$$\sum_{j\in\{1,\ldots,m\}} 2^{-\ell_j} \leqslant 1$$

Conversely, if the lengths ℓ_j obey Kraft's inequality, then there is a prefix code with such lengths.

We also know that $D(p\|q)\geq 0$, such that $E_p[\log 1/p(X)] \leq E_p[\log 1/q(X)]$ for all p,q.

In other words: $E_p[\log 1/q(X)]$ is minimized for q=p.

The Upper Bound

Let p be the probability distribution of variable X.
Try defining a prefix code C with $\ell_j = \lceil -\log p_j \rceil$.

Algebra tells us:

$$\sum_j 2^{-\ell_j} = \sum_j 2^{-\lceil -\log p_j \rceil} \leqslant \sum_j 2^{\log p_j} = 1$$

$$\boxed{\text{Kraft's inequality tells us that there does exist a prefix code C with such lengths } \ell_j.}$$

The expected length of this code is:

$E_p[\ell_j] = E_p[\lceil -\log p(X) \rceil] < E_p[-\log p(X)]+1 = H(X)+1.$

How to get rid of the annoying +1 term?

Lowering the Upper Bound

To reduce the $H(X)+1$ upper bound to $H(X)+\varepsilon$, we apply the same approach to strings in X^n instead of X.

The new probability distribution $p^{\otimes n}$ is defined by:
$p^{\otimes n}(x_1,\ldots,x_n) = p(x_1)\cdots p(x_n)$ such that $H(X^n) = n\cdot H(X)$.

We know that there exists a prefix code C_n for X^n with $L(C_n) < H(X^n) + 1$, hence the average per letter is now bounded by:

$$\frac{L(C_n)}{n} < \frac{H(X^n)+1}{n} = H(X) + \frac{1}{n}$$

By letting $n\to\infty$, we get $L(C_n)/n \to H(X)$.

Source Coding Algorithms

How to implement Shannon's idea for lossless source coding for a known distribution p(X)?

Two important cases:

$\rightarrow$ Huffman codes

$\rightarrow$ Shannon, Shannon-Fano, Shannon-Fano-Elias

An important but different setting occurs when we do not know a priori the distribution p.

$\rightarrow$ Lempel-Ziv compression

Optimal Coding?

Let $\mathcal{X}=\{0,1\}$ with $p(X=0) = 0.99$ and $p(X=1) = 0.01$.

Shannon's original idea suggests a code with
$\ell_0 = \lceil -\log p_0 \rceil = 1$ bit and $\ell_1 = \lceil -\log p_1 \rceil = 7$ bits,
which is clearly not optimal.

Looking at the code tree it is clear that for an optimal code the two longest words must have equal length and differ only in the last bits.

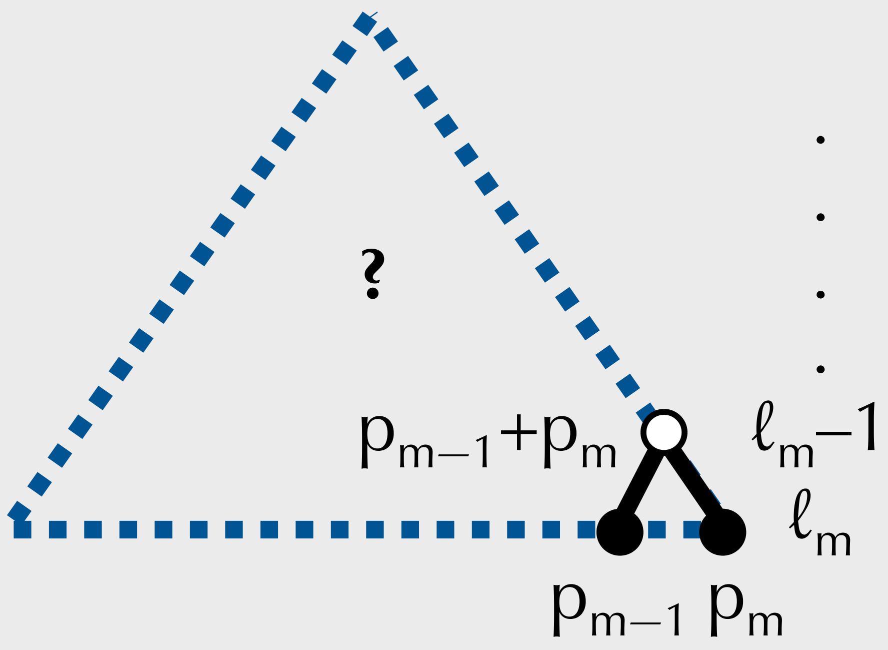
Another optimal property: if $p_j > p_k$ then $\ell_j \leq \ell_k$
(otherwise the $j \leftrightarrow k$ swapped code is better).

These observations lead to the Huffman code
(which is indeed optimal).

Properties of Optimal Codes

Assume probabilities $p_1 \geq \ldots \geq p_m$ for alphabet $\mathcal{X} = \{1,\ldots,m\}$.

- For an optimal code tree the two longest code words must have equal length and differ only in the last bit.
- For an optimal code it holds that if $p_j > p_k$ then $\ell_j \leq \ell_k$.
- Hence the two code words at depth $\ell_{max} = \ell_m$ correspond to the two smallest probabilities p_m and p_{m-1}.



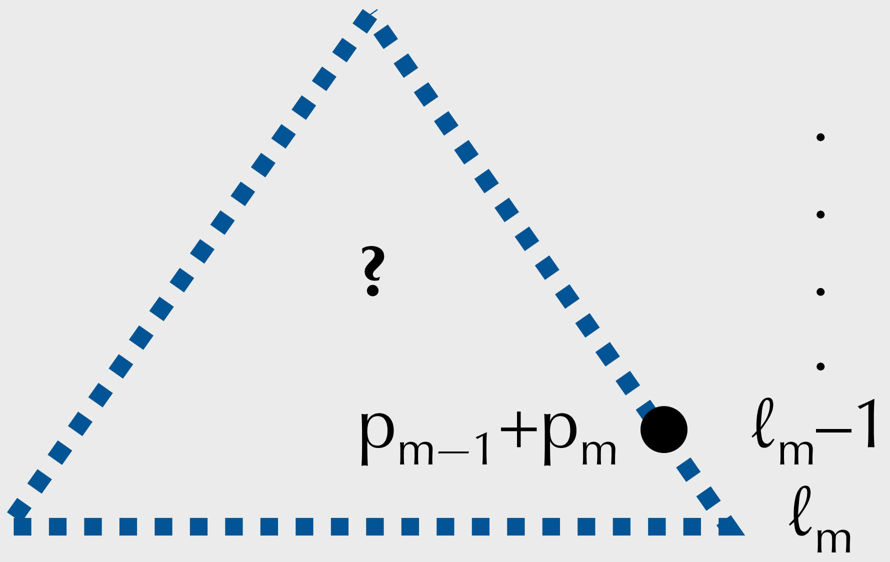
This show that $L(C) = \sum_j p_j \ell_j$ equals $p_1\ell_1 + p_2\ell_2 \ldots + (p_{m-1}+p_m)\ell_m$

$= p_1\ell_1 + p_2\ell_2 \ldots + (p_{m-1}+p_m)(\ell_m - 1) + p_{m-1} + p_m$

Recursive Definition

Assume probabilities $p_1,\ldots,p_m$ for alphabet $\mathcal{X} = \{1\ldots,m\}$.
Optimizing the m lengths $\ell_1,\ldots,\ell_m$ for the m probabilities $p_1,\ldots,p_m$ equals optimizing the m–1 lengths $\ell_1, \ell_2,\ldots, \ell_{m-1}$ for the m–1 probabilities $p_1, p_2,\ldots, p_{m-1}+p_m$.

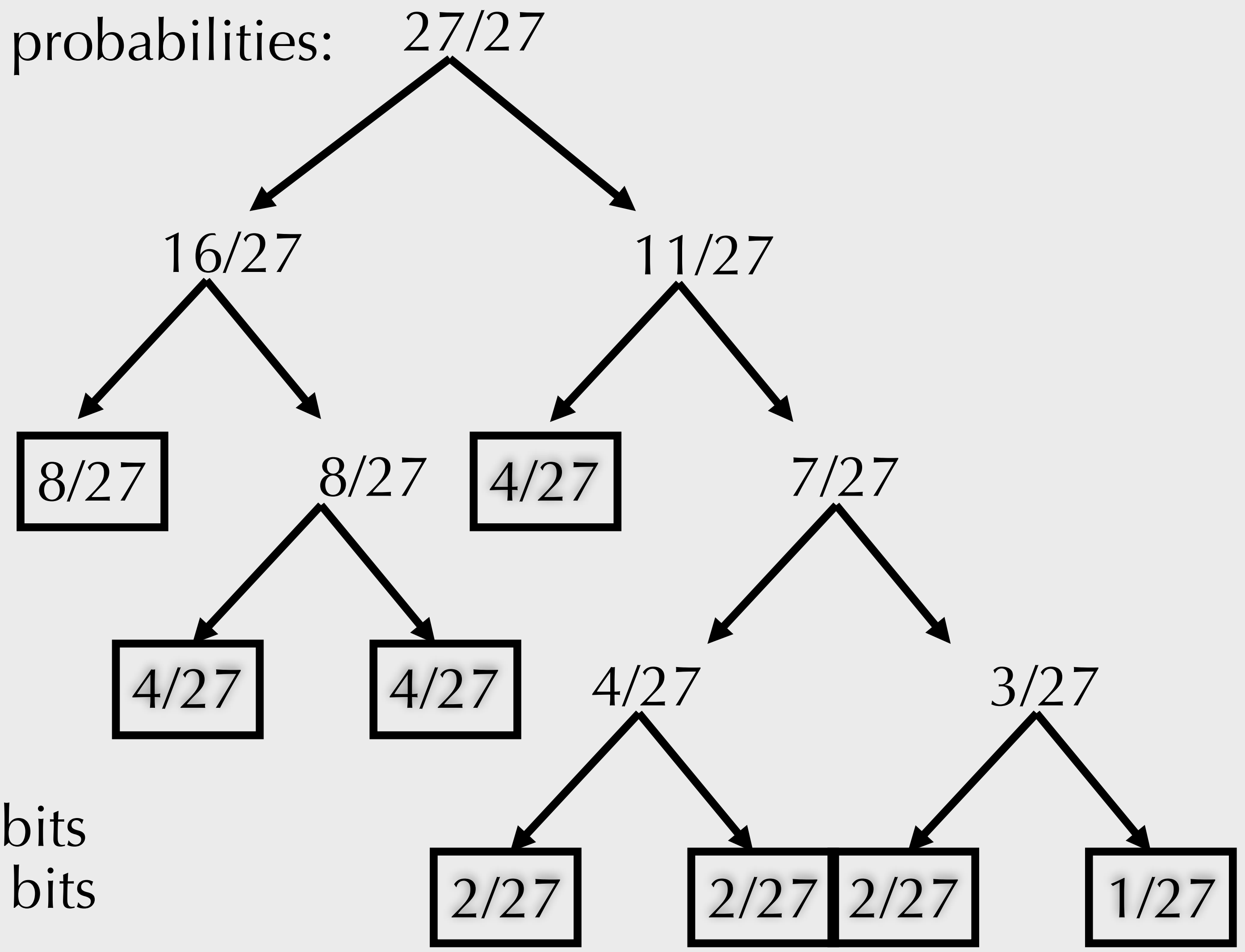
We apply the same line of reasoning for the smaller code tree: group the two smallest probabilities, et c.



Applying this approach recursively gives the Huffman code for $p_1,\ldots,p_m$.

Huffman Code Example

What optimal code tree do
we get for the probabilities:
1/27,
2/27,
2/27,
2/27,
4/27,
4/27,
4/27,
8/27?



L(C) = 2.815 bits
H(p) = 2.755 bits

Huffman Codes

Huffman codes were invented in 1951 by David Huffman as the result of a term paper, assigned by his MIT professor Robert Fano.

The bottom-up construction of Huffman codes is optimal, unlike Shannon-Fano codes, which are made top down.

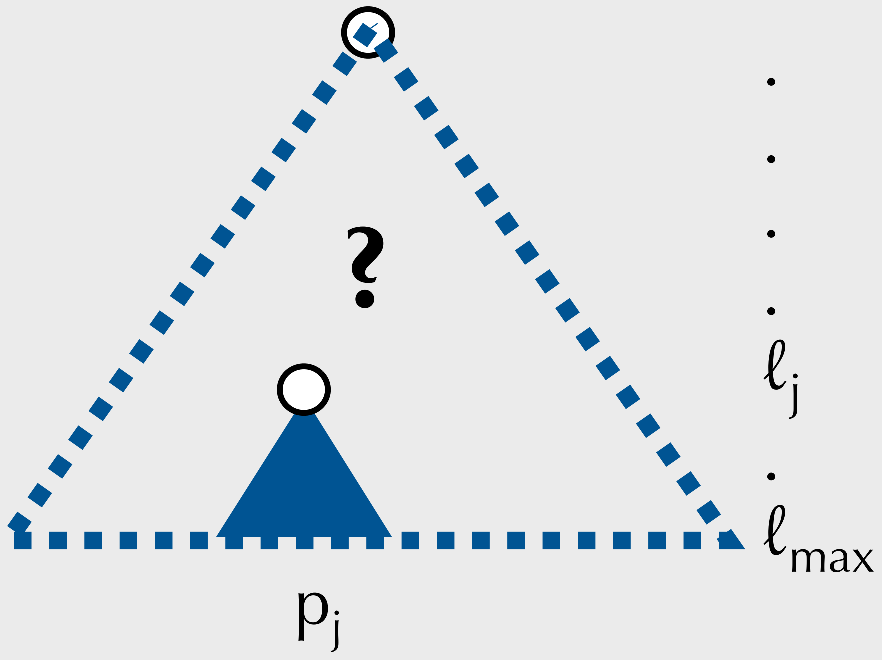
Key idea of Shannon-Fano codes: Divide the probabilities into two groups such that their weights are ≈(1/2,1/2).
At the root of the code tree hang two sub code trees for the two groups. Apply the same idea recursively.

Shannon-Fano-Elias Codes

The proof of Kraft's inequality uses the idea of a code tree embedded in a complete tree of depth ℓ_{max} such that the 'code leafs' act as roots of sub-trees with $2^{\ell_{max}-\ell_j}$ leafs at depth ℓ_{max} of the complete tree.

Key idea for Shannon-Fano-Elias codes: Optimal code trees have leafs at depth $\ell_j = -\log(p_j)$. As part of the complete tree, this gives $2^{\ell_{max}-\ell_j}$ leafs hanging under the node j.

As a fraction, this is p_j of all $2^{\ell_{max}}$ leafs of the complete tree.



Shannon-Fano-Elias Codes II

Building the code tree bottom-up:

- Line up the probabilities $p_1,\ldots,p_m$.

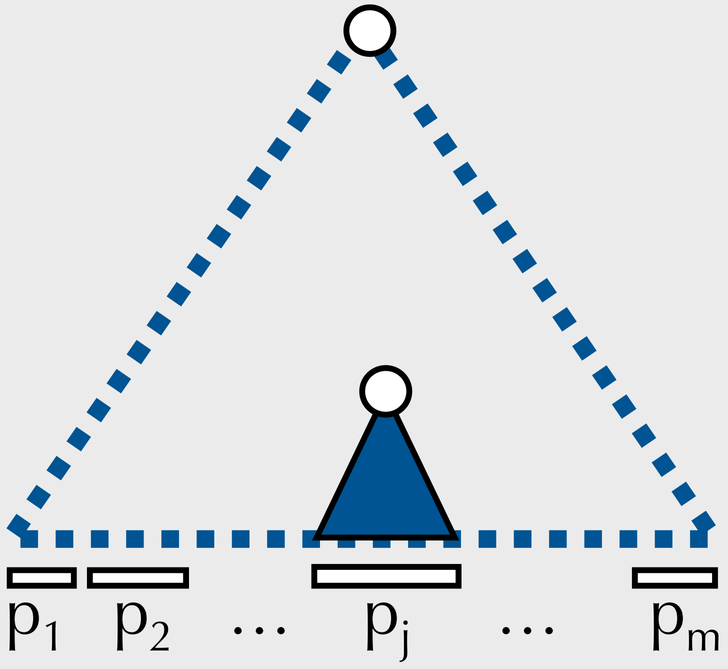
- Assign a number of leafs of the complete tree proportional to p_j.

- Build ‘upwards’ the subtree corresponding to the word j.

- Do this for all $1\leq j\leq m$.

If the subtrees would fit perfectly, then this construction would be optimal with $L(C)=H(p)$.

In general we have to adjust the construction to make it ‘fit’.



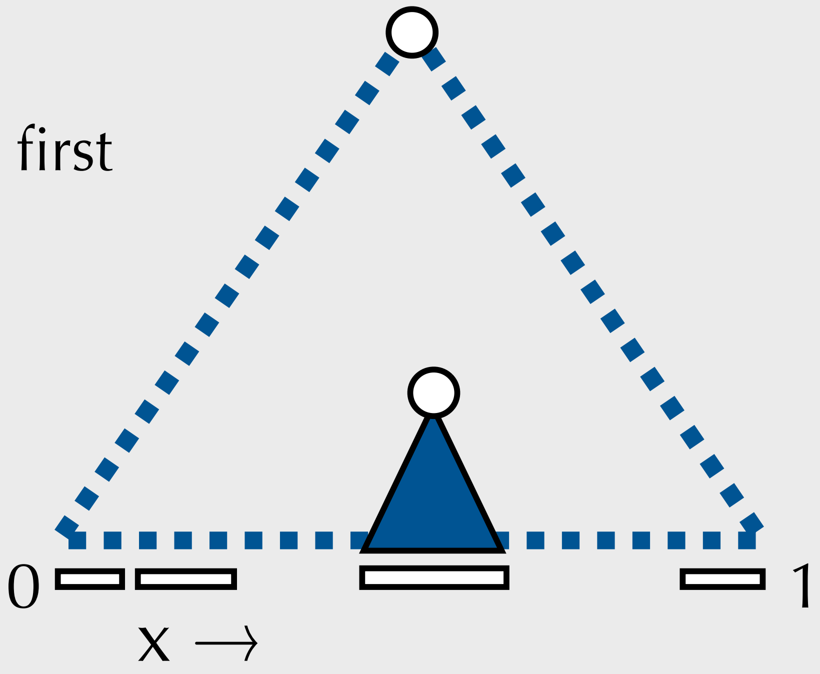
Shannon-Fano-Elias Codes III

Construction of a Shannon-Fano-Elias code for $p=(p_1,\ldots,p_m)$ and $\mathcal{X}=\{1,\ldots,m\}$ (all p_j are non-zero):

- Define the cumulative function: $F(t) = \sum_{j\leqslant t} p_j$.

- Adjust this function by $F'(t) = F(t) - \frac{1}{2} p_t$.
(The values $F'(j)$ indicates the x-coordinate of the node of the code word for j.)

- Write out $F'(j)$ in binary and use the first $\lceil -\log p_j \rceil + 1$ bits as the code leaf for j.



Example 5.9.2

Example Shannon-Fano-Elias code for $\mathcal{X}=\{1,\ldots,5\}$
with $p(1)=p(2)=1/4$, $p(3)=1/5$, $p(4)=p(5)=3/20$:

j	1	2	3	4	5
$p(j)$	0.25	0.25	0.2	0.15	0.15
$F(j)$	0.25	0.5	0.7	0.85	1
$F'(j)$	0.125	0.375	0.6	0.775	0.925
$F'(j)_2$	0.001	0.011	0.100110…	0.110001…	0.111011…
ℓ_j	3	3	4	4	4
c_j	001	011	1001	1100	1110

Note that this code with its $L(C) = 3.5$ is sub-optimal.
$H(p) = 2.285$, so according to Shannon's theorem
there is a better prefix code with $L(C) < 3.285$.

Length of SFE Codes

By the construction of the Shannon-Fano-Elias code we know that $\ell_j = \lceil -\log p_j \rceil + 1$ for all j, hence for $L(C) = \sum_j p_j \cdot \ell_j = \sum_j p_j \cdot (\lceil -\log p_j \rceil + 1) < \sum_j p_j \cdot (-\log p_j + 2) = H(p) + 2$.

The code is optimal up to 2 bits.

Why is this code a proper prefix code?

The code words $c_j \in \{0,1\}^*$ can be viewed as reals $0.c_j \in [0,1)$ (base 2 notation) that approximate the $F'(j) \in [0,1)$ from below.

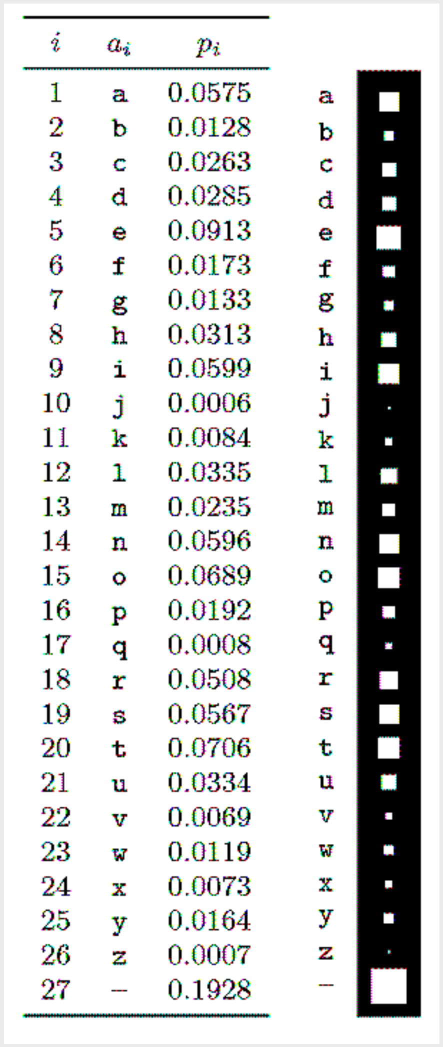
By construction we have $F'(j) - c_j < 2^{-\ell_j} \leq \frac{1}{2} p_j$.

The distance between $F'(j-1)$ and $F'(j)$ is $\frac{1}{2}(p_{j-1} + p_j)$, hence from c_j we can reliably recover j.

G. Perec's "La Disparition"

A Void's plot follows a group of individuals hunting a missing companion, Anton Vowl. It is in part a parody of noir and horror fiction, with many stylistic tricks and gags, plot convolutions, and a grim conclusion. In many parts it implicitly talks about its own lipogrammatic limitation: illustrations of this including missing subdivision 5 of 26, and calling its main protagonist "Vowl". Individuals within A Void do work out what is missing, but find difficulty discussing or naming it as it has no word or form that conforms to its author's constraint. Philip Howard, writing a lipogrammatic appraisal of A Void in a British daily journal, said, "This is a story chock-full of plots and sub-plots, of loops within loops, of trails in pursuit of trails, all of which allow its author an opportunity to display his customary virtuosity as an avant-gardist magician, acrobat and clown."

Letter Frequencies in English



i	a_i	p_i
1	a	0.0575
2	b	0.0128
3	c	0.0263
4	d	0.0285
5	e	0.0913
6	f	0.0173
7	g	0.0133
8	h	0.0313
9	i	0.0599
10	j	0.0006
11	k	0.0084
12	l	0.0335
13	m	0.0235
14	n	0.0596
15	o	0.0689
16	p	0.0192
17	q	0.0008
18	r	0.0508
19	s	0.0567
20	t	0.0706
21	u	0.0334
22	v	0.0069
23	w	0.0119
24	x	0.0073
25	y	0.0164
26	z	0.0007
27	–	0.1928

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
–

From *Information Theory, Inference and Learning Algorithms*, D. MacKay.

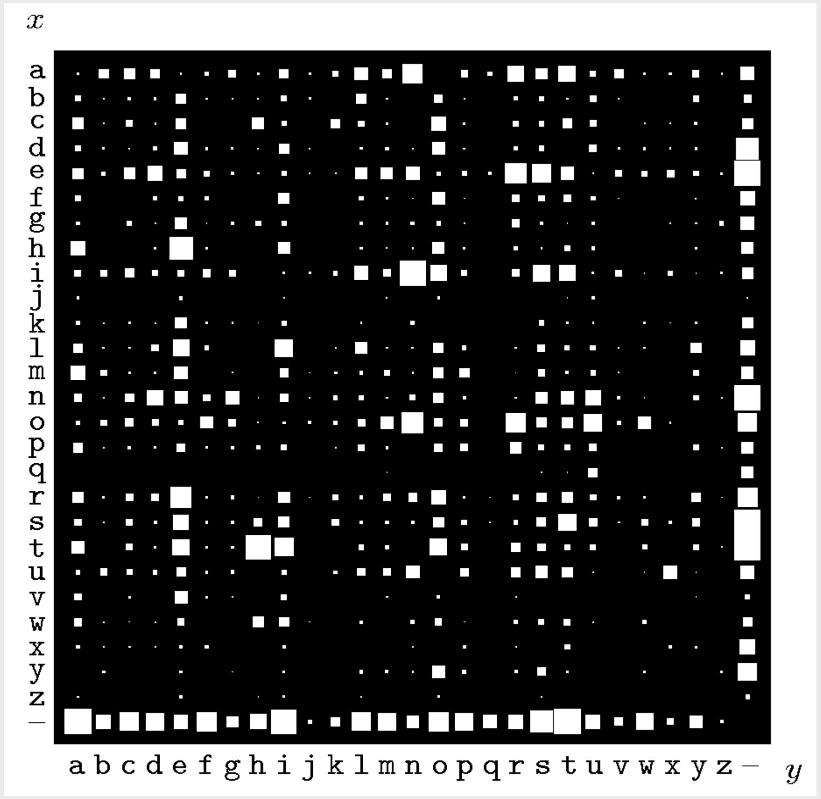
The 27 letter alphabet gives an upper bound on the entropy of log 27 = 4.75.

A random sentence: XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGHYD…

The different single letter frequencies gives a lower entropy of ≈4.1 bits/letter.

A typical sentence: “OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA….”

Combined Letter Frequencies



For letter pairs, the entropy is 4.03 bits per letter.

Typical sentence using triplets: "IN NO IST LAT WHEY CRATICT FROURE BIRS GROCID …"

For letter quadruplets, the entropy is down to 2.8 bits/letter.

What is the entropy of English?
See http://math.ucsd.edu/~crypto/java/ENTROPY/

Entropy of English?

See [EoIT, §6.4] for details.

Entropy is estimated by [Shannon'51] to be between 0.3 and 1.5 bits per letter. (Note log 27 = 4.75…)

The redundancy of English is useful when trying to break codes (like Shannon did in WWII).

Redundancy, Redundancy

The fact that a natural language is very redundant is very useful: it allows us to understand English texts that are written or spoken unclearly.

Redundancy of images is also helpful when we are reproducing them imperfectly (copying, faxing, et c.)

But when we are using reliable (digital) technology, then the natural redundancy incurs an unnecessary cost when transmitting information.

Shannon's idea while working for Bell Labs: Data Compression

Bell Laboratories
The Wisdom Business.

Universal Coding

We want to be able to compress strings without having to know a priori what the probabilities p(X) are. In other words, we want a compression algorithm that can be applied to any string, without requiring any additional information: universal source coding [EoIT, Chapter 13].

Two examples:

- Lempel-Ziv coding $\rightarrow$ [EoIT, §13.4–5]

- (Arithmetic coding $\rightarrow$ [EoIT, §13.3])

Lempel-Ziv Coding

Lempel-Ziv Coding [LZ78] works as follows [EoIT, §13.4]:

- Reading the string left to right, break it up such that each part is a previous part plus one additional bit.

For an n bit string, let there be c(n) parts $w_1,\ldots,w_{c(n)}$.

- Code w_k by (j,b) with a pointer $0 \leq j < k$ and the additional bit b such that $w_k = w_j b$. (The pointer requires log c(n) bits.)

Example: s = 1011010100010 broken up is 1,0,11,01,010,00,10.

Adding pointers: (0,1)(0,0)(1,1)(2,1)(4,0)(2,0)(1,0).

Writing out pointers in binary and removing punctuation, gives LZ(s) = 00010000001101011000001000010.

Efficiency of Lempel-Ziv

For binary strings of length n, the length of the LZ compressed string will be approximately $c(n)\cdot\lceil\log c(n) + 1\rceil$ bits.

One can show that in the limit for large n we have:
- For all possible strings X^n: $c(n) < n/\log n$
- More tightly: $c(n) \lesssim n\cdot H(X)/\log n$.

Hence, for large strings the LZ compression gives a string of length $\lesssim n\cdot H(X) + n\cdot H(X)/\log n$, which thus uses $\lesssim H(X) + H(X)/\log n \approx H(X)$ bits per letter.

Proving these limits is not straightforward;
see [EoIT, §13.5] for details.

Applications of LZ

The idea behind LZ78 compression is used many compression algorithms: LZW, compress (UNIX), gzip, DEFLATE, GIF, TIFF, PDF.