Data Compression

Arithmetic coding

Arithmetic Coding: Introduction

Allows using “fractional” parts of bits!!

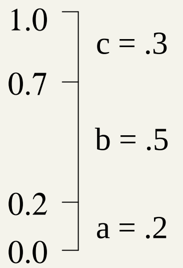
Used in PPM, JPEG/MPEG (as option), Bzip

More time costly than Huffman, but integer implementation is not too bad.

Arithmetic Coding (message intervals)

Assign each symbol to an interval range from 0 (inclusive) to 1 (exclusive).

e.g.



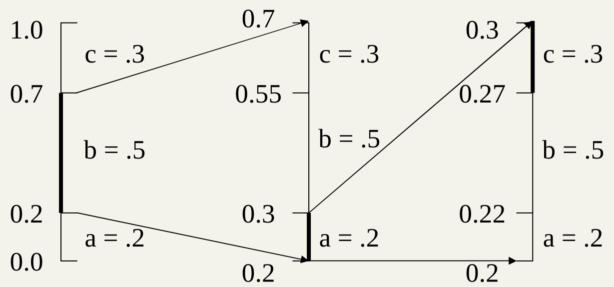
$$f(i) = \sum_{j=1}^{i-1} p(j)$$

$f(a) = .0,\quad f(b) = .2,\quad f(c) = .7$

The interval for a particular symbol will be called the <u>**symbol interval**</u> (e.g for b it is [.2,.7))

Arithmetic Coding: Encoding Example

Coding the message sequence: **bac**



The final sequence interval is **[.27,.3)**

Arithmetic Coding

To code a sequence of symbols c with probabilities p[c] use the following:

$$l_0 = 0 \quad l_i = l_{i-1} + s_{i-1} * f[c_i]$$

$$s_0 = 1 \quad s_i = s_{i-1} * p[c_i]$$

f[c] is the cumulative prob. up to symbol c (not included)

Final interval size is

$$s_n = \prod_{i=1}^{n} p[c_i]$$

The interval for a message sequence will be called the **sequence interval**

Uniquely defining an interval

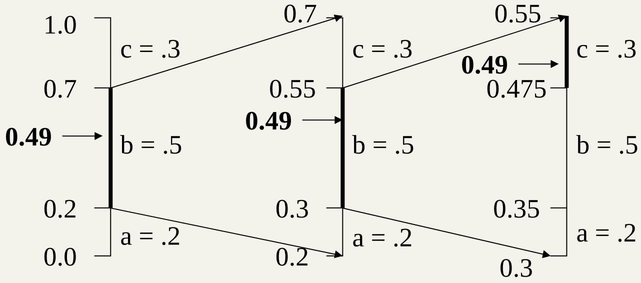
Important property: The intervals for distinct messages of length *n* will never overlap

Therefore by specifying any number in the final interval uniquely determines the msg.

Decoding is similar to encoding, but on each step need to determine what the message value is and then reduce interval

Arithmetic Coding: Decoding Example

Decoding the number .49, **knowing the message is of length 3**:



The message is **bbc.**

Representing a real number

Binary fractional representation:

$$.75 = .11$$
$$1/3 = .01\overline{01}$$
$$11/16 = .1011$$

Algorithm
1. **x = 2 *x**
2. **If x < 1 output 0**
3. **else x = x - 1; output 1**

So how about just using the **shortest** binary fractional representation in the sequence interval.

e.g. [0,.33) = **.01** [.33,.66) = **.1** [.66,1) = **.11**

Representing a code interval

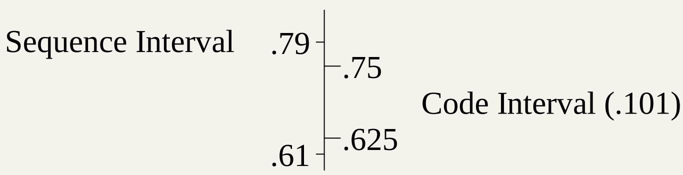
Can view binary fractional numbers as **intervals** by considering all completions.

	min	max	interval
.11	$.11\overline{0}$	$.11\overline{1}$	[.75,1.0)
.101	$.101\overline{0}$	$.101\overline{1}$	[.625,.75)

We will call this the **code interval.**

Selecting the code interval

To find a prefix code, find a binary fractional number whose code interval is contained in the sequence interval (*dyadic number*).



Can use L + s/2 truncated to 1 + $\lceil \log (1/s) \rceil$ bits

Bound on Arithmetic length

Lemma 3.20 *$\forall l, s \geq 0$ such that $l + s < 1$, the truncation of $l + \frac{s}{2}$ to the first $-\lfloor \log_2 s \rfloor + 1$ bits is in $[l, l+s)$.*

Proof: Clearly we have that $l + \frac{s}{2} \in [l, l+s)$, but we want to prove that also its truncation is in the interval. If we let $l + \frac{s}{2} = 0.b_1 b_2 \ldots$, it differs from its truncation to the first h bits $(0.b_1 \ldots b_h)$ of at most:

$$l + \frac{s}{2} - \text{trunc}_h(l + \frac{s}{2}) = \sum_{i=h+1}^{\infty} b_i \cdot 2^{-i} \leq 2^{-h} \sum_{i=1}^{\infty} 2^{-i} = 2^{-h}$$

So we have

$$l + \frac{s}{2} - 2^{-h} \leq \text{trunc}_h(l + \frac{s}{2}) \leq l + \frac{s}{2}$$

and if we let $h = -\lfloor \log_2 s \rfloor + 1$ we have

$$2^{-h} = 2^{\lfloor \log_2 s \rfloor - 1} \leq \frac{s}{2} \quad \text{and} \quad l \leq \text{trunc}_h(l + \frac{s}{2}) \leq l + \frac{s}{2}$$

so $\text{trunc}_h(l + \frac{s}{2}) \in [l, l+s)$. ■

Note that $-\lfloor \log s \rfloor + 1 = \lceil \log (2/s) \rceil$

Bound on Length

Theorem: *For a text of length n, the Arithmetic encoder generates at most*

$1 + \lceil \log (1/s) \rceil =$

$= 1 + \lceil \log \Pi (1/p_i) \rceil$

$\leq 2 + \Sigma_{j=1,n} \log (1/p_i)$

$= 2 + \Sigma_{k=1,|\Sigma|} np_k \log (1/p_k)$

$= 2 + n H_0$ bits

$nH_0 + 0.02\, n$ bits in practice because of rounding

Integer Arithmetic Coding

Problem is that operations on arbitrary precision real numbers is *expensive*.

Key Ideas of integer version:

- Keep integers in range [0..R) where $R=2^k$
- Use rounding to generate integer interval
- Whenever sequence intervals falls into **top**, **bottom** or **middle half**, expand the interval by a **factor 2**

Integer Arithmetic is an approximation

Integer Arithmetic (scaling)

If $l \geq R/2$ then (**top half**)
Output 1 followed by m 0s
$m = 0$
Message interval is expanded by 2

If $u < R/2$ then (**bottom half**)
Output 0 followed by m 1s
$m = 0$
Message interval is expanded by 2

If $l \geq R/4$ and $u < 3R/4$ then (**middle half)**
Increment m
Message interval is expanded by 2

All other cases, just continue...

You find this at

Range encoding - Wikipedia, the free encyclopedia - Mozilla Firefox

http://en.wikipedia.org/wiki/Range_encoding

Range encoding

From Wikipedia, the free encyclopedia

Range encoding is a form of arithmetic coding, a data compression method, that is believed to be free from arithmetic coding related patents. It is on this basis that interest in range encoding has arisen, particularly in the open source community. However, it is often observed that range encoding appears to be just a slightly different description of arithmetic coding, and that the two are actually the same. On this matter, it may be noted that while range *encoding*, as defined by G N N Martin in his 1979 paper on "Range encoding: an algorithm for removing redundancy from a digitised message",[1] is essentially the same as arithmetic coding, range *encoders* often use a particular kind of implementation based on Martin's paper. It is such implementations that are believed to be free from arithmetic coding related patents, on the basis of the age of Martin's paper.

Contents [hide]

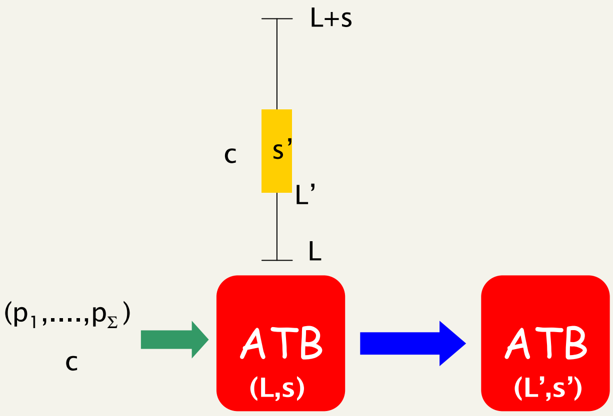
How range encoding works

Range encoding conceptually encodes all the symbols of the message into one number, unlike Huffman coding which assigns each symbol a bit-pattern and concatenates all the bit-patterns together. Thus range encoding can achieve greater compression ratios than the one-bit-per-symbol upper bound on Huffman encoding and it does not suffer the inefficiencies that Huffman does when dealing with probabilities that are not exact powers of two.

The central concept behind range encoding is this: given a large-enough range of integers, and a probability estimation for the symbols, the initial range can easily be divided into sub-ranges whose sizes are proportional to the probability of the symbol they represent. Each symbol of the message can then be encoded in turn, by reducing the current range down to just that sub-range which corresponds to the next symbol to

Arithmetic ToolBox

As a state machine



Therefore, even the distribution can change over time

K-th order models: PPM

Use previous *k* characters as the context.

- Makes use of conditional probabilities
- This is the *changing* distribution

Base probabilities on counts:

e.g. if seen **th** 12 times followed by **e** 7 times, then the conditional probability *p(**e**|**th**)* = 7/12.

Need to keep *k* small so that dictionary does not get too large (typically less than 8).

PPM: Partial Matching

Problem: What do we do if we have not seen context followed by character before?

- Cannot code 0 probabilities!

The key idea of PPM is to reduce context size if previous match has not been seen.

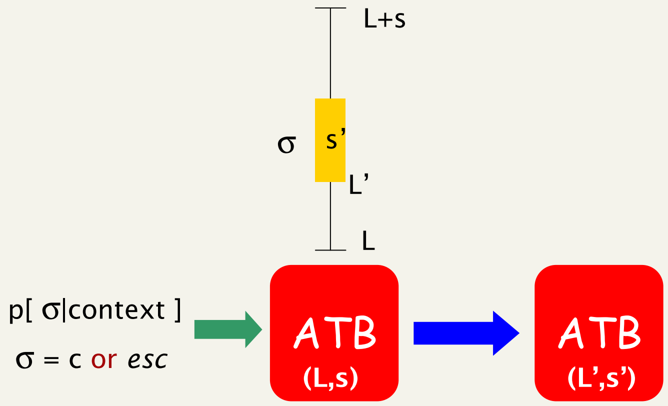
- If character has not been seen before with current context of size 3, send an *escape-msg* and then try context of size 2, and then again an *escape-msg* and context of size 1, ….

Keep statistics for each context size < *k*

The escape is a special character with some probability.

- Different variants of PPM use different heuristics for the probability.

PPM + Arithmetic ToolBox



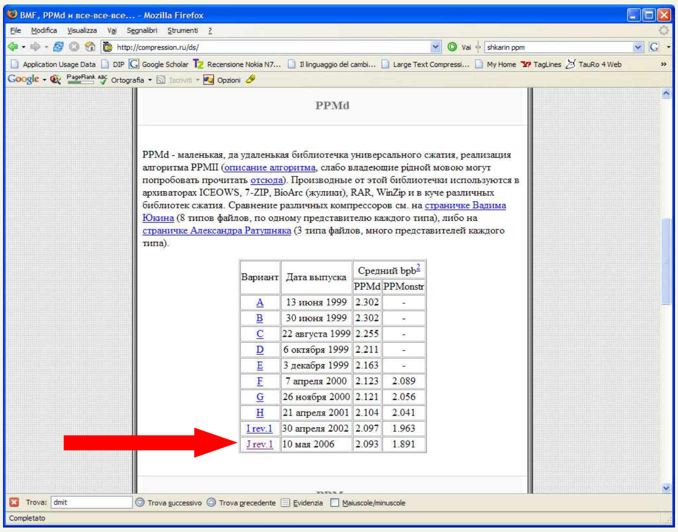
Encoder and Decoder must know the *protocol* for selecting the *same* conditional probability distribution (PPM-variant)

PPM: Example Contexts

Context	Counts	Context	Counts	Context	Counts
Empty	A = 4	A	C = 3	AC	B = 1
	B = 2		$ = 1		C = 2
	C = 5	B	A = 2		$ = 2
	$ = 3		$ = 1	BA	C = 1
		C	A = 1		$ = 1
			B = 2	CA	C = 1
			C = 2		$ = 1
			$ = 3	CB	A = 2
					$ = 1
				CC	A = 1
					B = 1
					$ = 2

String = ACCBACCACBA **B** k = 2

You find this at: compression.ru/ds/



PPMd

PPMd - маленькая, да удаленькая библиотечка универсального сжатия, реализация алгоритма PPMII (описание алгоритма, слабо владеющие рiдной мовою могут попробовать прочитать отсюда). Производные от этой библиотечки используются в архиваторах ICEOWS, 7-ZIP, BioArc (жулики), RAR, WinZip и в куче различных библиотек сжатия. Сравнение различных компрессоров см. на страничке Вадима Юкина (8 типов файлов, по одному представителю каждого типа), либо на страничке Александра Ратушняка (3 типа файлов, много представителей каждого типа).

Вариант	Дата выпуска	Средний bpb[2]	
		PPMd	PPMonstr
A	13 июня 1999	2.302	-
B	30 июня 1999	2.302	-
C	22 августа 1999	2.255	-
D	6 октября 1999	2.211	-
E	3 декабря 1999	2.163	-
F	7 апреля 2000	2.123	2.089
G	26 ноября 2000	2.121	2.056
H	21 апреля 2001	2.104	2.041
I rev.1	30 апреля 2002	2.097	1.963
J rev.1	10 мая 2006	2.093	1.891