15-583:Algorithms in the Real World

Data Compression II

Arithmetic Coding

– Integer implementation

Applications of Probability Coding

– Run length coding: Fax ITU T4

– Residual coding: JPEG-LS

– Context Coding: PPM

Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, …

Information Theory: Entropy, etc.

Probability Coding: Huffman + Arithmetic Coding

Applications of Probability Coding: PPM + others

Lempel-Ziv Algorithms: LZ77, gzip, compress, ...

Other Lossless Algorithms: Burrows-Wheeler

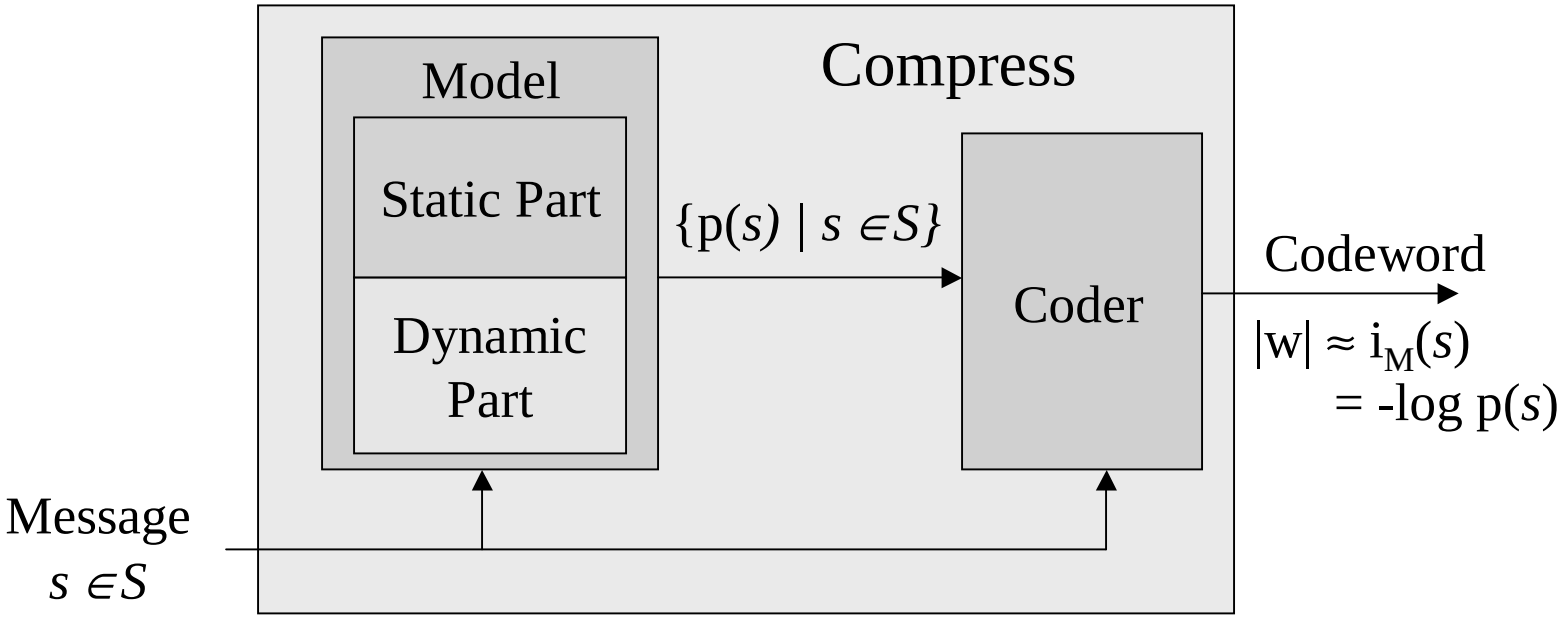
Lossy algorithms for images: JPEG, fractals, ...

Lossy algorithms for sound?: MP3, ...

Key points from last lecture

- **Model** generates probabilities, **Coder** uses them
- **Probabilities** are related to **information**. The more you know, the less info a message will give.
- More "skew" in probabilities gives lower **Entropy H** and therefore better compression
- **Context** can help "skew" probabilities (lower H)
- Average length l_a for **optimal prefix code** bound by $H \leq l_a < H + 1$
- **Huffman codes** are optimal prefix codes

Encoding: Model and Coder

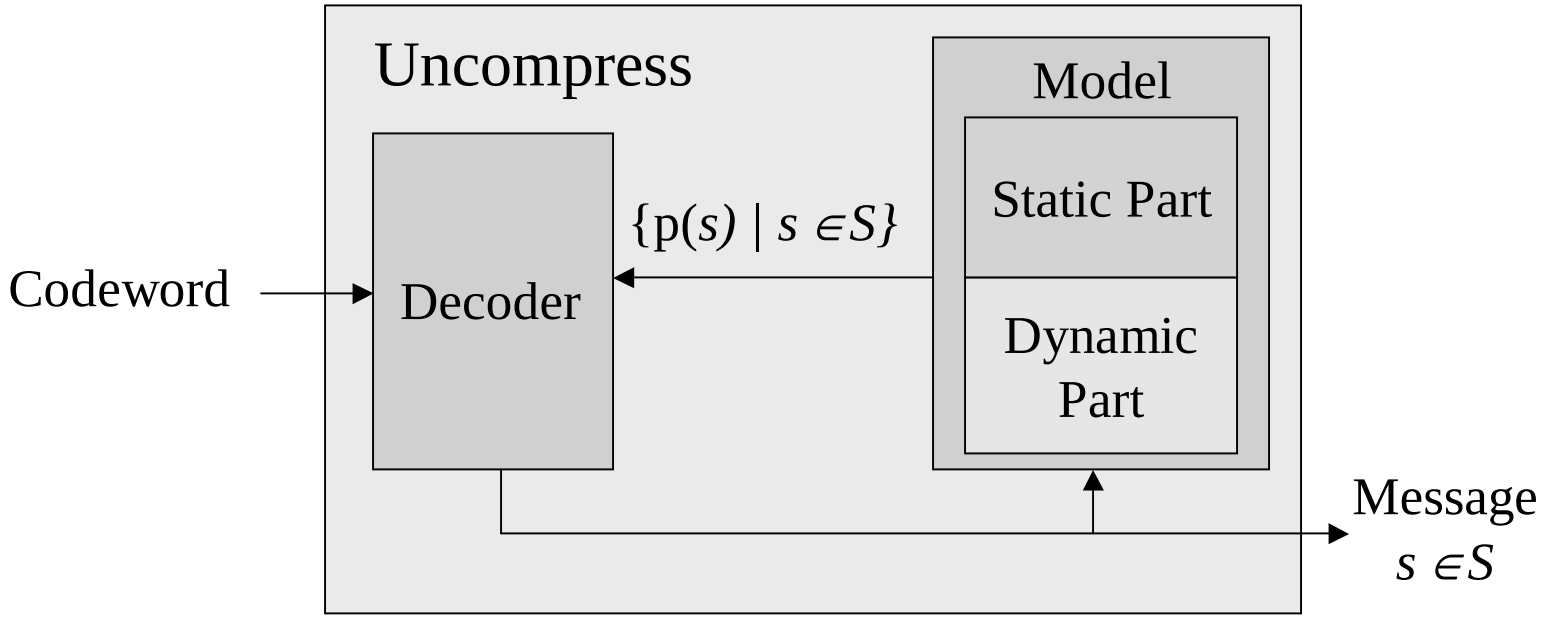


- The **Static part** of the model is fixed
- The **Dynamic part** is based on previous messages

The "optimality" of the code is relative to the probabilities.

If they are not accurate, the code is not going to be efficient

Decoding: Model and Decoder



The **probabilities** {p(s) | s ∈ S} generated by the model need to be the same as generated in the encoder.

Note: consecutive "messages" can be from a different message sets, and the probability distribution can change

Adaptive Huffman Codes

Huffman codes can be made to be adaptive without completely recalculating the tree on each step.

- Can account for changing probabilities
- Small changes in probability, typically make small changes to the Huffman tree

Used frequently in practice

Review some definitions

Message: an atomic unit that we will code.

- Comes from a message set $S = \{s_1,\ldots,s_n\}$ with a probability distribution $p(s)$. Probabilities must sum to 1. Set can be infinite.
- Also called symbol or character.

Message sequence: a sequence of messages, possibly each from its own probability distribution

Code *C(s)*: A mapping from a message set to **codewords**, each of which is a string of bits

Problem with Huffman Coding

Consider a message with probability .999. The self information of this message is

$$-\log(.999) = .00144$$

If we were to send a 1000 such message we might hope to use 1000*.0014 = 1.44 bits.

Using Huffman codes we require at least one bit per message, so we would require 1000 bits.

Arithmetic Coding: Introduction

Allows "blending" of bits in a message sequence. Only requires 3 bits for the example

Can bound total bits required based on sum of self information:

$$l < 2 + \sum_{i=1}^{n} s_i$$

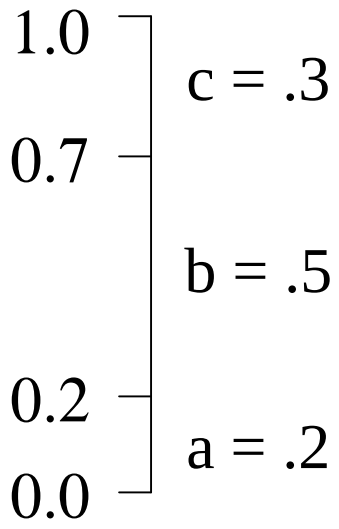
Used in PPM, JPEG/MPEG (as option), DMM

More expensive than Huffman coding, but integer implementation is not too bad.

Arithmetic Coding (message intervals)

Assign each probability distribution 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 message will be called the **message interval** (e.g for b the interval is [.2,.7))

Arithmetic Coding (sequence intervals)

To code a message use the following:

$$l_1 = f_1 \quad l_i = l_{i-1} + s_{i-1} f_i$$
$$s_1 = p_1 \quad s_i = s_{i-1} p_i$$

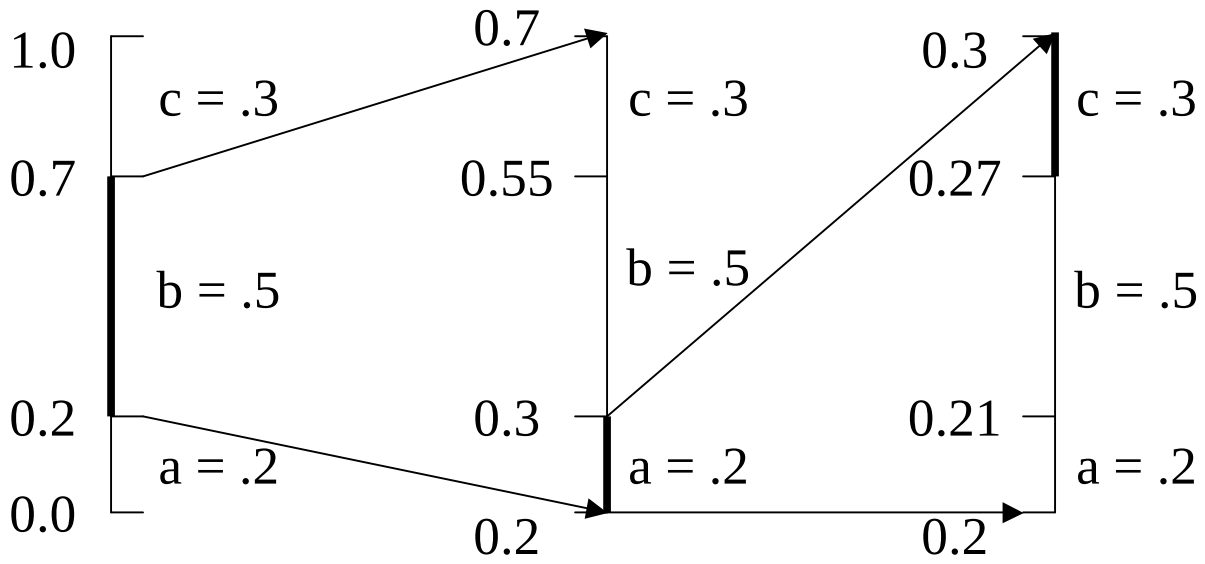
Each message narrows the interval by a factor of p_i.

Final interval size: $$s_n = \prod_{i=1}^{n} p_i$$

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

Arithmetic Coding: Encoding Example

Coding the message sequence: **bac**



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

Uniquely defining an interval

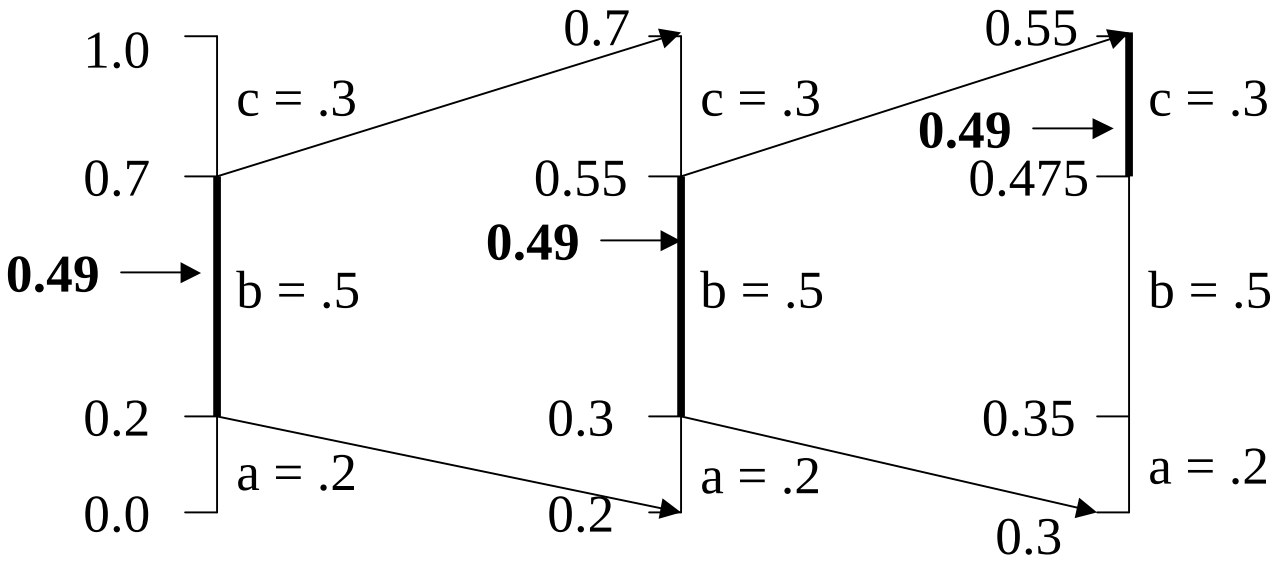
Important property: The sequence intervals for distinct message sequences of length *n* will never overlap

Therefore: specifying any number in the final interval uniquely determines the sequence.

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 an Interval

Binary fractional representation:

$.75 = .11$

$1/3 = .0\overline{01}$

$11/16 = .1011$

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

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

But what if you receive a 1?

Is the code complete? (Not a prefix code)

Representing an Interval (continued)

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

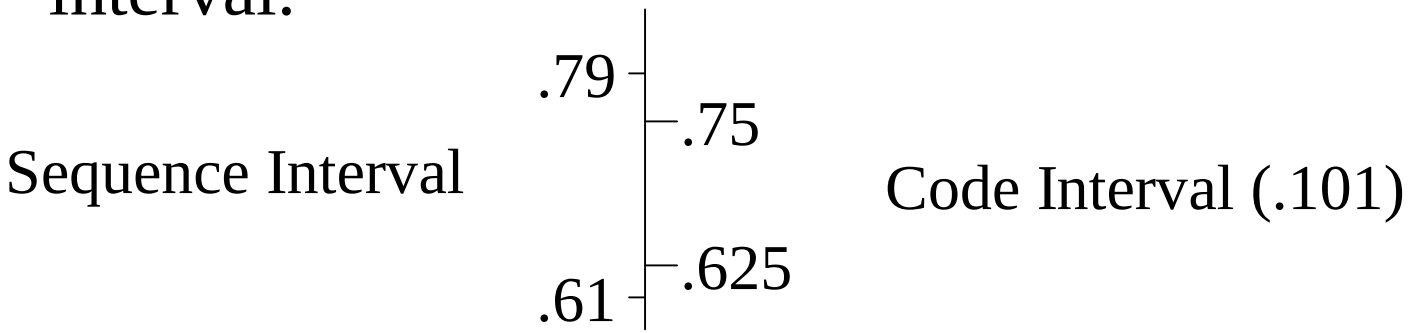
	min	max	interval
.11	$.11\overline{0}$	$.11\overline{1}$	[.75,1.0)
.101	$.101\overline{0}$	$.101\overline{1}$	[.625,.75)

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

Lemma: *If a set of code intervals do not overlap then the corresponding codes form a prefix code.*

Selecting the Code Interval

To find a prefix code find a binary fractional number whose code interval is contained in the sequence interval.



e.g. [0,.33) = .00 [.33,.66) = .100 [.66,1) = .11

Can use *l* + *s/2* truncated to

bits $\lceil -\log(s/2) \rceil = 1 + \lceil -\log s \rceil$

RealArith Encoding and Decoding

RealArithEncode:

- Determine *l* and *s* using original recurrences
- Code using $l + s/2$ truncated to $1+\lceil -\log s \rceil$ bits

RealArithDecode:

- Read bits as needed so code interval falls within a message interval, and then narrow sequence interval.
- Repeat until *n* messages have been decoded .

Bound on Length

Theorem: *For n messages with self information* $\{s_1,\ldots,s_n\}$ RealArithEncode *will generate at most* $2+\sum_{i=1}^{n} s_i$ *bits.*

$$
\begin{aligned}
1+\lceil -\log s \rceil &= 1+\left\lceil -\log\left(\prod_{i=1}^{n} p_i\right)\right\rceil \\
&= 1+\left\lceil \sum_{i=1}^{n} -\log p_i \right\rceil \\
&= 1+\left\lceil \sum_{i=1}^{n} s_i \right\rceil \\
&< 2+\sum_{i=1}^{n} s_i
\end{aligned}
$$

Integer Arithmetic Coding

Problem with RealArithCode 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 factor of 2 Integer Algorithm is an approximation

Integer Arithmetic Coding

The probability distribution as integers

- Probabilities as counts:
  e.g. c(1) = 11, c(2) = 7, c(3) = 30
- S is the sum of counts
  e.g. 48 (11+7+30)
- Partial sums f as before:
  e.g. f(1) = 0, f(2) = 11, f(3) = 18

Require that R > 4S so that probabilities do not get rounded to zero

Integer Arithmetic (contracting)

$$\boldsymbol{l_1 = 0,\ \ s_1 = R}$$

$$s_i = u_i - l_i + 1$$

$$u_i = l_i + \left\lfloor s_i \cdot (f_i + s_i) / T \right\rfloor - 1$$

$$l_i = l_i + \left\lfloor s_i \cdot f_i / T \right\rfloor$$

Integer Arithmetic (scaling)

If $l \geq R/2$ then (**in top half**)

Output 1 followed by m 0s

$m = 0$

Scale message interval by expanding by 2

If $u < R/2$ then (**in bottom half**)

Output 0 followed by m 1s

$m = 0$

Scale message interval by expanding by 2

If $l \geq R/4$ and $u < 3R/4$ then (**in middle half)**

Increment m

Scale message interval by expanding by 2

Applications of Probability Coding

How do we generate the probabilities?

Using character frequencies directly does not work very well (e.g. 4.5 bits/char for text).

Technique 1: transforming the data

- Run length coding (ITU Fax standard)
- Move-to-front coding (Used in Burrows-Wheeler)
- Residual coding (JPEG LS)

Technique 2: using conditional probabilities

- Fixed context (JBIG…almost)
- Partial matching (PPM)

Run Length Coding

Code by specifying message value followed by number of repeated values:

e.g. **abbbaacccca => (a,1),(b,3),(a,2),(c,4),(a,1)**

The characters and counts can be coded based on frequency.

This allows for small number of bits overhead for low counts such as 1.

Facsimile ITU T4 (Group 3)

Standard used by all home Fax Machines

ITU = International Telecommunications Standard

Run length encodes sequences of black+white pixels

Fixed Huffman Code for all documents. e.g.

Run length	White	Black
1	000111	010
2	0111	11
10	00111	0000100

Since alternate black and white, no need for values.

Move to Front Coding

Transforms message sequence into sequence of integers, that can then be probability coded

Start with values in a total order:
e.g.: [a,b,c,d,e,….]

For each message output position in the order and then move to the front of the order.
e.g.: b => output: 3, new order: [c,a,b,d,e,…]
a => output: 2, new order: [a,c,b,d,e,…]

Codes well if there are concentrations of message values in the message sequence.

Residual Coding

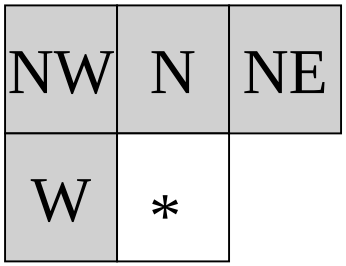
Used for message values with meaningfull order
e.g. integers or floats.

Basic Idea: guess next value based on current context. Output difference between guess and actual value. Use probability code on the output.

JPEG-LS

JPEG Lossless (not to be confused with lossless JPEG)
Just completed standardization process.

Codes in Raster Order. Uses 4 pixels as context:



Tries to guess value of * based on W, NW, N and NE.

Works in two stages

JPEG LS: Stage 1

Uses the following equation:

$$P = \begin{cases} \min(N,W) & \text{if } NW \geq \max(N,W) \\ \max(N,W) & \text{if } NW < \min(N,W) \\ N + W - NW & \text{otherwise} \end{cases}$$

Averages neighbors and captures edges. e.g.

40	3	*
40	3	

30	40	*
20	30	

3	3	*
40	40	

JPEG LS: Stage 2

Uses 3 gradients: W-NW, NW-N, N-NE

- Classifies each into one of 9 categories.
- This gives 9^3=729 contexts, of which only 365 are needed because of symmetry.
- Each context has a bias term that is used to adjust the previous prediction

After correction, the residual between guessed and actual value is found and coded using a Golomblike code.

Using Conditional Probabilities: PPM

Use previous *k* characters as the context.

Base probabilities on counts:

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

Need to keep *k* small so that dictionary does not get too large.

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

– Cannot code 0 probabilities!

PPM: Partial Matching

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, try context of size 2, and then context of size 1, and then no context

Keep statistics for each context size < *k*

PPM: Changing between context

How do we tell the decoder to use a smaller context?

Send an **escape** message. Each escape tells the decoder to reduce the size of the context by 1.

The escape can be viewed as special character, but needs to be assigned a probability.

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

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	A = 1
			C = 2		$ = 1
			$ = 3	CB	A = 2
					$ = 1
				CC	A = 1
					B = 1
					$ = 2

String = ACCBACCACBA k = 2

PPM: Other important optimizations

If context has not been seen before, automatically escape (no need for an escape symbol since decoder knows previous contexts)

Can exclude certain possibilities when switching down a context. This can save 20% in final length!