15-853:Algorithms in the Real World

Data Compression: Lecture 3

Summary so far

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)

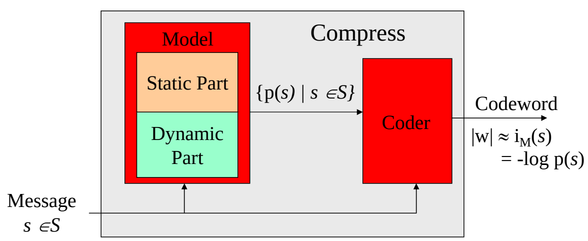
Average length l_a for **optimal prefix code** bound by

$$H \le l_a < H + 1$$

Huffman codes are optimal prefix codes

Arithmetic codes allow "blending" among messages

Encoding: Model and Coder



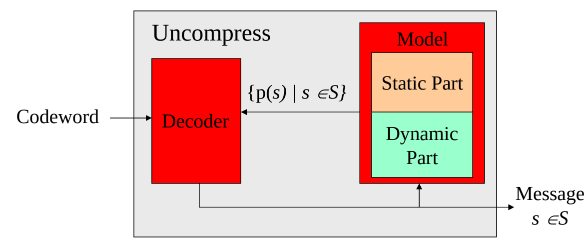
The **Static part** of the model is fixed

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** $\{p(s) \mid s \in S\}$ 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

Codes with Dynamic Probabilities

Huffman codes:

Need to generate a new tree for new probabilities.

Small changes in probability, typically make small changes to the Huffman tree.

"**Adaptive Huffman codes**" update the tree without having to completely recalculate it.

Used frequently in practice

Arithmetic codes:

Need to recalculate the f(m) values based on current probabilities.

Can be done with a balanced tree.

Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, …

Information Theory: Entropy, etc.

Probability Coding: Huffman + Arithmetic Coding

➡ **Applications of Probability Coding**: PPM + others

- Transform coding: move to front, run-length, …
- Context coding: fixed context, partial matching

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

Other Lossless Algorithms: Burrows-Wheeler

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

Compressing graphs and meshes: BBK

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 the 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

Takes advantage of **temporal locality**

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

For each message

- output the position in the order
- move to the front of the order.

e.g.: c => output: 3, new order: [c,a,b,d,e,…]
a => output: 2, new order: [a,c,b,d,e,…]

Probability code the output.

The hope is that there is a bias for small numbers.

Residual Coding

Typically used for message values that represent some sort of amplitude:
e.g. gray-level in an image, or amplitude in audio.

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)

Codes in Raster Order. Uses 4 pixels as context:

NW	N	NE
W	*	

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 Golomb-like code. (Golomb codes are similar to Gamma codes)

Using Conditional Probabilities: PPM

Use previous *k* characters as the context.

- Makes use of conditional probabilities

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).

Note that 8-gram Entropy of English is about 2.3bits/char while PPM does as well as 1.7bits/char

PPM: Partial Matching

Problem: What do we do if we have not seen the context followed by the 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, 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!

It is critical to use arithmetic codes since the probabilities are small.

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,
- LZ78, compress (Not covered in class)

Other Lossless Algorithms: Burrows-Wheeler

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

Compressing graphs and meshes: BBK

Lempel-Ziv Algorithms

LZ77 (Sliding Window)

Variants: LZSS (Lempel-Ziv-Storer-Szymanski)

Applications: `gzip`, Squeeze, LHA, PKZIP, ZOO

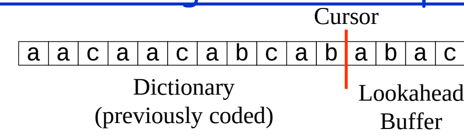
LZ78 (Dictionary Based)

Variants: LZW (Lempel-Ziv-Welch), LZC

Applications: `compress`, GIF, CCITT (modems), ARC, PAK

Traditionally LZ77 was better but slower, but the gzip version is almost as fast as any LZ78.

LZ77: Sliding Window Lempel-Ziv



Dictionary and **buffer** "windows" are fixed length and slide with the **cursor**

Repeat:

Output (***p***, ***l***, **c**) where

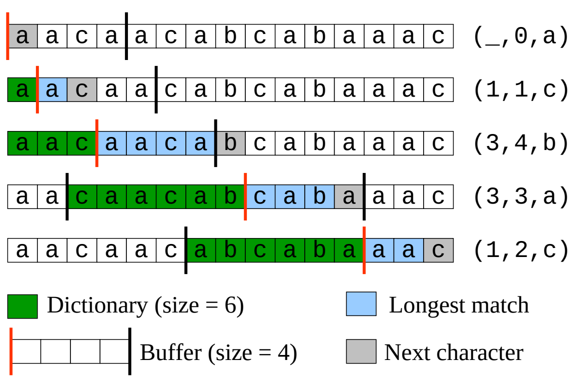
p = position of the longest match in the dictionary (relative to the cursor)

l = length of longest match

c = next char in buffer beyond longest match

Advance window by ***l*** + 1

LZ77: Example



LZ77 Decoding

Decoder keeps same dictionary window as encoder.

For each message it looks it up in the dictionary and inserts a copy

What if *l* > p? (only part of the message is in the dictionary.)

E.g. dict = `abcd`, codeword = `(2,9,e)`

- Simply copy from left to right

```
for (i = 0; i < length; i++)
  out[cursor+i] = out[cursor-offset+i]
```

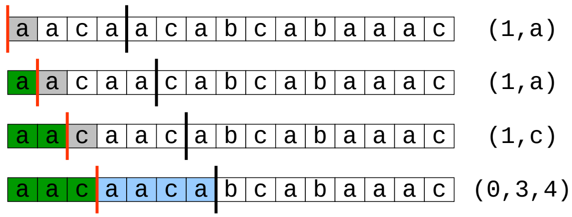
- Out = `abcdcdcdcdcdce`

LZ77 Optimizations used by gzip

LZSS: Output one of the following two formats

(0, position, length) or **(1,char)**

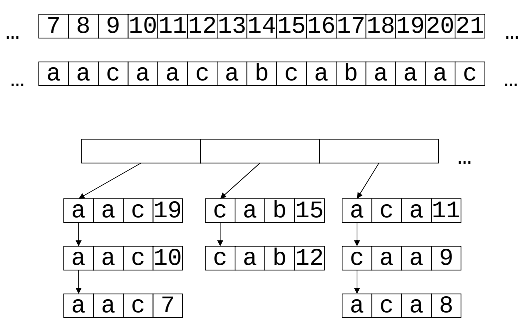
Uses the second format if length < 3.



Optimizations used by gzip (cont.)

1. Huffman code the positions, lengths and chars
2. Non greedy: possibly use shorter match so that next match is better
3. Use a hash table to store the dictionary.
   - Hash keys are all strings of length 3 in the dictionary window.
   - Find the longest match within the correct hash bucket.
   - Puts a limit on the length of the search within a bucket.
   - Within each bucket store in order of position

The Hash Table



Theory behind LZ77

Sliding Window LZ is Asymptotically Optimal [Wyner-Ziv,94]

Will compress long enough strings to the source entropy as the window size goes to infinity.

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \to \infty} H_n$$

Uses logarithmic code (e.g. gamma) for the position.

Problem: “long enough” is really **really** long.

Lempel-Ziv Algorithms Summary

Both LZ77 and LZ78 and their variants keep a "dictionary" of recent strings that have been seen.

The differences are:

- How the dictionary is stored (LZ78 is a trie)
- How it is extended (LZ78 only extends an existing entry by one character)
- How it is indexed (LZ78 indexes the nodes of the trie)
- How elements are removed

Lempel-Ziv Algorithms Summary (II)

Adapts well to changes in the file (e.g. a Tar file with many file types within it).

Initial algorithms did not use probability coding and performed poorly in terms of compression. More modern versions (e.g. gzip) do use probability coding as "second pass" and compress much better.

The algorithms are becoming outdated, but ideas are used in many of the newer algorithms.

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
- ACB

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

Compressing graphs and meshes: BBK

Burrows -Wheeler

Currently near best "balanced" algorithm for text

Breaks file into fixed-size blocks and encodes each block separately.

For each block:

- Sort each character by its full context. This is called the **block sorting transform**.
- Use **move-to-front transform** to encode the sorted characters.

The ingenious observation is that the decoder only needs the sorted characters and a pointer to the first character of the original sequence.

Burrows Wheeler: Example

Let's encode: $\mathbf{d_1e_2c_3o_4d_5e_6}$

We've numbered the characters to distinguish them.

Context "wraps" around. Last char is most significant.

Context	Char
$ecode_6$	d_1
$coded_1$	e_2
$odede_2$	c_3
$dedec_3$	o_4
$edeco_4$	d_5
$decod_5$	e_6

Sort Context →

Context	Output
$dedec_3$	o_4
$coded_1$	e_2
$decod_5$	e_6
$odede_2$	c_3
$ecode_6$	d_1 ⇐
$edeco_4$	d_5

Burrows-Wheeler (Continued)

Theorem: *After sorting, equal valued characters appear in the same order in the output as in the most significant position of the context.*

Proof sketch: Since the chars have equal value in the most-significant-position of the context, they will be ordered by the rest of the context, *i.e.* the previous chars. This is also the order of the output since it is sorted by the previous characters.

Context	Output
[redacted]c_3	o_4
[redacted]d_1	e_2
[redacted]d_5	e_6
[redacted]e_2	c_3
[redacted]$\mathbf{e}_6$	d_1
[redacted]$\mathbf{o}_4$	d_5

Burrows-Wheeler: Decoding

Consider dropping all but the last character of the context.

- What follows the underlined **a** ?
- What follows the underlined **b**?
- What is the whole string?

Context	***Output***
a	c
a	**b**
a	b
b	a
b	**a** ⇐
c	a

Answer: `b, a, abacab`

Burrows-Wheeler: Decoding

What about now?

Answer: `cabbaa`

Can also use the "rank".

The "rank" is the position of a character if it were sorted using a stable sort.

Context	***Output***	***Rank***
a	c ⇐	6
a	a	1
a	b	4
b	b	5
b	a	2
c	a	3

Burrows-Wheeler Decode

Function BW_Decode(In, Start, n)
  S = MoveToFrontDecode(In,n)
  R = Rank(*S*)
  j = Start
  for *i=1* **to** *n* **do**
    Out[*i*] = S[*j*]
    j = *R*[*j*]

Rank gives position of each char in sorted order.

Decode Example

S	Rank(S)
o_4	6
e_2	4
e_6	5
c_3	1
d_1	2
d_5	3

	Out
e_6	d_1 ⇐
d_1	e_2
e_2	c_3
c_3	o_4
o_4	d_5
d_5	e_6

Overview of Text Compression

PPM and **Burrows-Wheeler** both encode a single character based on the immediately preceding context.

LZ77 and **LZ78** encode multiple characters based on matches found in a block of preceding text

Can you **mix these ideas**, *i.e.,* code multiple characters based on immediately preceding context?

- **BZ** does this, but they don't give details on how it works – current best compressor
- **ACB** also does this – close to best

ACB (Associate Coder of Buyanovsky)

Keep dictionary sorted by context (the last character is the most significant)

- Find longest match for context
- Find longest match for contents
- Code
  - Distance between matches in the sorted order
  - Length of contents match

Has aspects of Burrows-Wheeler, and LZ77

Context	Contents
	decode
dec	ode
d	ecode
decod	e
de	code
deco	de