15-853: Algorithms in the Real World

Announcement:
No recitation this week.

Scribe Volunteer?
Recap

**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.

**Arithmetic codes** allow “blending” among messages.
Recap: Exploiting context

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)
## Recap: Integer codes (detour)

<table>
<thead>
<tr>
<th>n</th>
<th>Binary</th>
<th>Unary</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>..001</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>..010</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>..011</td>
<td>110</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>..100</td>
<td>1110</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>..101</td>
<td>11110</td>
<td>110</td>
</tr>
<tr>
<td>6</td>
<td>..110</td>
<td>111110</td>
<td>110</td>
</tr>
</tbody>
</table>

Many other fixed prefix codes:
- Golomb, phased-binary, subexponential, ...
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)
Recap: Run Length Coding

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

e.g. \textit{abbbaacccca} => (a,1),(b,3),(a,2),(c,4),(a,1)

The characters and counts can be coded based on frequency (i.e., probability coding).

Q: Why?

Typically low counts such as 1 and 2 are more common => use small number of bits overhead for these.

Used as a sub-step in many compression algorithms.
Reap: Move to Front Coding

- Transforms message sequence into sequence of integers
- Then probability code
- 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 a**
  
  c => output: 3, new order: [c,a,b,d,e,...]
  a => output: 2, new order: [a,c,b,d,e,...]

Used as a sub-step in many compression algorithms.
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.

E.g.: Consider compressing a stock value over time.

Residual coding is used in JPEG Lossless
JPEG-LS

JPEG Lossless
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.

The residual between guessed and actual value is found and then coded using a Golomb-like code.
(Golomb codes are similar to Gamma codes)
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)
  - in reading notes
- Partial matching (PPM)
PPM: PREDICTION BY PARTIAL MATCHING
PPM: Using Conditional Probabilities

Makes use of conditional probabilities

- Use previous $k$ characters as context.

- Base probabilities on counts
  e.g. if seen $\text{th}$ 12 times and followed by $\text{e}$ 7 times, then the conditional probability of $\text{e}$ given $\text{th}$ is?

$$p(\text{e}|\text{th}) = \frac{7}{12}.$$

Each context has its own probability distribution

**Probability distribution will keep changing:**

Q: Is this a problem?

Fine as long as context precedes the character being coded since decoder knows the context
PPM example contexts

For context length \( k = 2 \)

<table>
<thead>
<tr>
<th>Context</th>
<th>Counts</th>
</tr>
</thead>
</table>
| AC      | B = 1  
          | C = 2  |
| BA      | C = 1  |
| CA      | C = 1  |
| CB      | A = 2  |
| CC      | A = 1  
          | B = 1  |

String = ACCBACCCACBA \( k = 2 \)
PPM: Challenges

**Challenge 1:** Dictionary size can get very large

Ideas?

- Need to keep $k$ small so that dictionary does not get too large
- Typically less than 8

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

**Challenge 2**: What do we do if we have not seen the context followed by the character before?

– Cannot code 0 probabilities!

E.g.: Say k=3. Have seen “cod” but not “code”. When ‘e’ appears what to do?

**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 (“ode”), and then context of size 1 (“de”), and then no context (“e”)

Keep statistics for each context size $< k$
### PPM: Example Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>A = 4</td>
<td>A</td>
<td>C = 3</td>
<td>AC</td>
<td>B = 1</td>
</tr>
<tr>
<td></td>
<td>B = 2</td>
<td>B</td>
<td>A = 2</td>
<td>BA</td>
<td>C = 2</td>
</tr>
<tr>
<td></td>
<td>C = 5</td>
<td>C</td>
<td>A = 1</td>
<td>CA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B = 2</td>
<td>CB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C = 2</td>
<td>CC</td>
<td></td>
</tr>
</tbody>
</table>

String = ACCBACCACBA

k = 2

To code “B” next?
PPM: Changing between context

Q: 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.
### PPM: Example Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>A = 4</td>
<td>A</td>
<td>C = 3</td>
<td>AC</td>
<td>B = 1</td>
</tr>
<tr>
<td></td>
<td>B = 2</td>
<td>B</td>
<td>A = 2</td>
<td>BA</td>
<td>C = 2</td>
</tr>
<tr>
<td></td>
<td>C = 5</td>
<td>C</td>
<td>C = 1</td>
<td>CA</td>
<td>C = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A = 1</td>
<td>CB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CC</td>
<td></td>
</tr>
</tbody>
</table>

String = ACCBACCACBAN

k = 2

To code “B” next?
PPM: Changing between context

Q: 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.

One option that works well in practice:

assign count = number of different characters seen (PPMC)
## PPM: Example Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>A = 4</td>
<td>A</td>
<td>C = 3</td>
<td>AC</td>
<td>B = 1</td>
</tr>
<tr>
<td></td>
<td>B = 2</td>
<td>B</td>
<td>$ = 1</td>
<td>C = 2</td>
<td>C = 2</td>
</tr>
<tr>
<td></td>
<td>C = 5</td>
<td>C</td>
<td>A = 2</td>
<td>$ = 2</td>
<td>$ = 2</td>
</tr>
<tr>
<td></td>
<td>$ = 3</td>
<td></td>
<td>$ = 1</td>
<td>C = 1</td>
<td>$ = 1</td>
</tr>
</tbody>
</table>

String = ACCBACCCACBA  \hspace{1cm} k = 2
PPM: Other important optimizations

Q: Do we always need multiple escapes when skipping multiple contexts?

If context has not been seen before, automatically escape
(no need additional escape symbol since decoder knows
previous contexts)
PPM: Optimizations example

<table>
<thead>
<tr>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>A = 4</td>
<td>A</td>
<td>C = 3</td>
<td>AC</td>
<td>B = 1</td>
</tr>
<tr>
<td></td>
<td>B = 2</td>
<td></td>
<td>$ = 1</td>
<td>C = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C = 5</td>
<td></td>
<td>A = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ = 3</td>
<td></td>
<td>$ = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$ = 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

String = ACCBACCACBACBA  

To code “A” next...  

k = 2
PPM: Other important optimizations

Q: Any other idea comes to mind?

Can exclude certain possibilities when switching down a context. This can save 20% in final length!
# PPM: Optimizations example

<table>
<thead>
<tr>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
<th>Context</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>A = 4</td>
<td>A</td>
<td>C = 3</td>
<td>AC</td>
<td>B = 1</td>
</tr>
<tr>
<td></td>
<td>B = 2</td>
<td></td>
<td>$ = 1</td>
<td>C = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C = 5</td>
<td></td>
<td>A = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$ = 3</td>
<td>B</td>
<td>$ = 1</td>
<td>$ = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>C</td>
<td>$ = 1</td>
<td>C = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A = 1</td>
<td>$ = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$ = 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C</td>
<td>$ = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$ = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$ = 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

String = ACCBACCACBACBA

k = 2

To code “A” next...
Q: Which probability code to use and why?

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

PPM: one of the best in terms of compression ratio but slow

We will soon learn about other techniques which come close to PPM but are way faster.
Compression Outline

**Introduction**: Lossy vs. Lossless, prefix codes, ...

**Information Theory**: Entropy, bounds on length, ...

**Probability Coding**: Huffman, Arithmetic Coding

**Applications of Probability Coding**: Run-length, Move-to-front, Residual, PPM

**Lempel-Ziv Algorithms**:
- LZ77, gzip,
- LZ78, compress (Not covered in class)
Lempel-Ziv Algorithms

Dictionary-based approach

Codes groups of characters at a time (unlike PPM)

High level idea:
- Look for longest match in the preceding text for the string starting at the current position
- Output a code for that string
- Move past the match
- Repeat
Lempel-Ziv Variants

**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 that starts 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

\[
\begin{array}{cccccccccccccccccccccccc}
\text{a} & \text{a} & \text{c} & \text{a} & \text{a} & \text{a} & \text{c} & \text{b} & \text{c} & \text{a} & \text{b} & \text{a} & \text{a} & \text{a} & \text{c} \\
(\_, 0, a) \\
\text{a} & \text{a} & \text{c} & \text{a} & \text{a} & \text{a} & \text{c} & \text{b} & \text{c} & \text{a} & \text{b} & \text{a} & \text{a} & \text{a} & \text{c} \\
(1, 1, c) \\
\text{a} & \text{a} & \text{c} & \text{a} & \text{a} & \text{a} & \text{c} & \text{b} & \text{c} & \text{a} & \text{b} & \text{a} & \text{a} & \text{a} & \text{c} \\
(3, 4, b) \\
\text{a} & \text{a} & \text{c} & \text{a} & \text{a} & \text{a} & \text{c} & \text{b} & \text{c} & \text{a} & \text{b} & \text{a} & \text{a} & \text{a} & \text{c} \\
(3, 3, a) \\
\text{a} & \text{a} & \text{c} & \text{a} & \text{a} & \text{a} & \text{c} & \text{b} & \text{c} & \text{a} & \text{b} & \text{a} & \text{a} & \text{a} & \text{c} \\
(1, 2, c) \\
\end{array}
\]

- Dictionary (size = 6)
- Longest match
- Buffer (size = 4)
- Next character
LZ77 Decoding

Decoder keeps same dictionary window as encoder. For each message it looks it up in the dictionary and inserts a copy at the end of the string.

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)

- Huffman code the positions, length and chars
- Non greedy: possibly use shorter match so that next match is better
- To quickly access the dictionary: Uses a hash table
  - Hash keys: every string of length 3
    - Why 3?
  - Find the longest match within the hash bucket with a fixed limit on length
  - Within each bucket store in order of position (helps select more recent match)
    - Why?
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.
Comparison to Lempel-Ziv 78

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

Lempel-Ziv-Welch variant in the reading notes
Lempel-Ziv Algorithms Summary

**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, prefix codes, ...

**Information Theory**: Entropy, bounds on length, ...

**Probability Coding**: Huffman, Arithmetic Coding

**Applications of Probability Coding**: Run-length, Move-to-front, Residual, PPM

**Lempel-Ziv Algorithms**:
- LZ77, gzip,
- LZ78, compress (Not covered in class)

**Other Lossless Algorithms**:
- Burrows-Wheeler
BURROWS-WHEELER
Burrows-Wheeler

Currently near best algorithm for text
Used in bzip2, genomics, ...

Transform coding technique (that has indirect connections to conditional probability techniques)

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

**For each block:**
- Create full context for each character (wraps around)
- Reverse lexical sort each character by its full context. This is called the “block sorting transform”.
- Use move-to-front transform on the sorted characters.
**Burrows Wheeler: Example**

To encode: \(d_1e_2c_3o_4d_5e_6\)

(Numbered the characters to distinguish them.)

Context “wraps” around. Last char is most significant.

<table>
<thead>
<tr>
<th>Context</th>
<th>Char</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecode_6</td>
<td>d_1</td>
<td>dedec_3</td>
<td>o_4</td>
</tr>
<tr>
<td>coded_1</td>
<td>e_2</td>
<td>coded_1</td>
<td>e_2</td>
</tr>
<tr>
<td>odede_2</td>
<td>c_3</td>
<td>decod_5</td>
<td>e_6</td>
</tr>
<tr>
<td>dedec_3</td>
<td>o_4</td>
<td>odede_2</td>
<td>c_3</td>
</tr>
<tr>
<td>edeco_4</td>
<td>d_5</td>
<td>edeco_4</td>
<td>d_5</td>
</tr>
<tr>
<td>decod_5</td>
<td>e_6</td>
<td>ecode_6</td>
<td>d_1</td>
</tr>
</tbody>
</table>

Q: Why is the output more easier to compress?
Burrows Wheeler: Example

<table>
<thead>
<tr>
<th>Context</th>
<th>Char</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecode₆</td>
<td>d₁</td>
<td>dedec₃</td>
<td>o₄</td>
</tr>
<tr>
<td>coded₁</td>
<td>e₂</td>
<td>coded₁</td>
<td>e₂</td>
</tr>
<tr>
<td>odede₂</td>
<td>c₃</td>
<td>decod₅</td>
<td>e₆</td>
</tr>
<tr>
<td>dedec₃</td>
<td>o₄</td>
<td>odede₂</td>
<td>c₃</td>
</tr>
<tr>
<td>edeco₄</td>
<td>d₅</td>
<td>ecode₆</td>
<td>d₁</td>
</tr>
<tr>
<td>decod₅</td>
<td>e₆</td>
<td>edeco₄</td>
<td>d₅</td>
</tr>
</tbody>
</table>

**Sort Context**

Gets similar characters together
(because we are ordering by context)

Can be viewed as giving a dynamically sized context.
(overcoming the problem of choosing the right “k” in PPM)

Why not just sort?
Can we invert BW Transform?

Output

\[ o_4 \]
\[ e_2 \]
\[ e_6 \]
\[ c_3 \]
\[ d_1 \leftarrow \]
\[ d_5 \]
Can we invert BW Transform?

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₃</td>
<td>o₄</td>
</tr>
<tr>
<td>d₁</td>
<td>e₂</td>
</tr>
<tr>
<td>d₅</td>
<td>e₆</td>
</tr>
<tr>
<td>e₂</td>
<td>c₃</td>
</tr>
<tr>
<td>e₆</td>
<td>d₁ ⇐</td>
</tr>
<tr>
<td>o₄</td>
<td>d₅</td>
</tr>
</tbody>
</table>

How can we get the last column of the context column from the output column?

Sort!

Any problem? Equal valued chars
Burrows-Wheeler (Continued)

**Theorem:** After sorting, equal valued characters appear in the same order in the output column as in the last column of the sorted context.

**Proof sketch:** ?

Since the chars with equal value in the most-significant-position (i.e., last column) 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.
Burrows-Wheeler: Decoding

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

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$c$</td>
</tr>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>$b$</td>
<td>$a$</td>
</tr>
<tr>
<td>$b$</td>
<td>$a$</td>
</tr>
</tbody>
</table>

Answer: b, a, abacab
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.

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>a c</td>
<td>cabbaa</td>
<td>6</td>
</tr>
<tr>
<td>a a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a b</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>b b</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>b a</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c a</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>