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.

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

\[ f(a) = 0.0, \quad f(b) = 0.2, \quad f(c) = 0.7 \]

The interval for a particular symbol will be called the **symbol interval** (e.g. for b it is [0.2, 0.7])

Arithmetic Coding: Encoding Example

Coding the message sequence: bac

The final sequence interval is [0.27, 0.3]
Arithmetic Coding

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

$$\begin{align*}
l_0 &= 0 \\
l_i &= l_{i-1} + s_{i-1} \cdot f[c_i] \\
s_0 &= 1 \\
s_i &= s_{i-1} \cdot p[c_i]
\end{align*}$$

$f[c]$ 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.
Decoding the number .49, knowing the message is of length 3:

The message is **bbc**.

### Representing a real number

**Binary fractional representation:**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary Fractional</th>
</tr>
</thead>
<tbody>
<tr>
<td>.75</td>
<td>.11</td>
</tr>
<tr>
<td>1/3</td>
<td>.0101</td>
</tr>
<tr>
<td>11/16</td>
<td>.1011</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>min</th>
<th>max</th>
<th>interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>.11</td>
<td>.110</td>
<td>[.75,1.0)</td>
</tr>
<tr>
<td>.101</td>
<td>.1010</td>
<td>[.625,.75)</td>
</tr>
</tbody>
</table>

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**).

Sequence Interval | .79 | .75 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.61</td>
<td>.625</td>
</tr>
</tbody>
</table>

**Can use L + s/2 truncated to 1 + ⌈log (1/s)⌉ 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 \(-[\log_2 s] + 1\) 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 \( l + \frac{s}{2} = 0.b_1b_2 \ldots \), 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 \( h = -[\log_2 s] + 1 \) we have
\[
 2^{-h} = 2^{-[\log_2 s]-1} \leq \frac{s}{2} \quad \text{and} \quad l \leq \text{trunc}_h(l + \frac{s}{2}) \leq l + \frac{s}{2}
\]
so \( \text{trunc}_h(l + \frac{s}{2}) \in [l, l + s) \). 

Note that \(-[\log_2 s] + 1 = \lceil \log (2/s) \rceil\)

Bound on Length

**Theorem:** For a text of length \( n \), the Arithmetic encoder generates at most
\[
 l + \lceil \log (1/s) \rceil = 1 + \lceil \log \prod (1/p_i) \rceil 
\]
\[
 \leq 2 + \sum_{j=1,n} \log (1/p_j) 
\]
\[
 = 2 + \sum_{k=1,|\Sigma|} np_k \log (1/p_k) 
\]
\[
 = 2 + n H_0 \text{ bits}
\]
\[
 nH_0 + 0.02 n \text{ bits in practice because of rounding}
\]
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

Arithmetic ToolBox

As a state machine

\[(p_1, ..., p_k, c) \rightarrow_{c} (L, s) \rightarrow_{L+s} (L', s')\]

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

$\sigma = c \text{ or } \text{esc}$

$\mathcal{P}(\sigma | \text{context })$

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

PPM: Example Contexts

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

String = ACCBACCACBA  \( k = 2 \)
You find this at: compression.ru/ds/