CE 108. S00
Introduction to Binary Arithmetic Coding

Glen G. Langdon, Jr.
Department of Computer Engineering
University of California
Santa Cruz, California

©1999, Glen G. Langdon, Jr.
Chapter 11
Binary Arithmetic Coding

Introduction

Outline

1. Arithmetic Coding
2. Binary version of Arithmetic Coding
3. Pasco’s solution to the precision problem, and example.
4. Advantages and Disadvantages of Arithmetic Coding for Compression

Advantages of Binary Arithmetic Codes

• The binary case is easily parameterized: let skew number \( k = -\log_2 p(LPS) \). With MPS value (1 bit) and less than 16 values of \( k \) (4-bits) you have a 5-bit parameter for the distribution.

• A binary arithmetic code is universal for any alphabet size by decomposing an \( m \)-ary alphabet into \( m-1 \) binary decisions.

• With parameterized implementation, the context model needs only provide the encoder with the 5-bit parameter and the binary value to be encoded. The decoder only needs the 5-bit parameter for decoding each binary value.

• Binary distributions are easily learned by counting techniques and what is learned can be converted to the 5-bit parameter for encoding.

• The JPEG lossless version with arithmetic coding employs the above ideas, first embodied in an algorithm called Sunset.

• A disadvantage relative to Huffman coding is that to code 8-bit symbols one requires (without pipelining) 8 encoder/decoder cycles.

Advantages of Arithmetic Coding in General

1. Coding can use the probabilities directly, provided the ordering is known. In \( m \)-ary case, it is relatively simple to use counting techniques in combination with arithmetic coding for one-pass adaptive compression. The count ratios are used in the encoding and decoding in determining the products \( A(s, b) = \frac{c(b)}{c(t|d)} \times A(s) \), and

\[
\text{Augend}(s, b) = \left( \frac{\sum_{a \in \mathcal{L}} c(a)}{c(t|d)} \right) \times A(s).
\]

2. With arithmetic coding, the exact probabilities can be represented to the precision the user is willing to provide.

3. Arithmetic codes, driven by probabilities themselves, perform better in the one-pass adaptive case than the current adaptive Huffman codes (version of Faller–Gallager).

4. In higher-order models, the code design convenience of arithmetic codes seems preferred over the design (and table storage) of many Huffman codes (one for each conditional probability).

Disadvantages of Arithmetic Coding in General

1. Decoding is a magnitude order search, which takes more time than a table look-up for Huffman codes.

2. In cases with the probability of the most probable symbol \( p(MPS) \) is still very small (the symbol alphabet is large), a major advantage of arithmetic coding’s ability to more accurately represent probabilities is lost.

• Little coding efficiency is to be gained by representing a small probability, between, say, \( 2^{-9} \) and \( 2^{-10} \), more accurately than an integer length code word.
Graphical Description of Recursive Version for Encoding.

Initial state: Current Interval: [0, 1)

- $C(A) = 0.0$: Current left end-point of Interval.
- $A(A) = 1.0$: Current Available space, or Width.
- $p(a)$ and $p(b)$ resp. mean probability of symbol $a$ and $b$.
- $P(b)$ is cumulative probability for $b$, where

$$P(b) = \sum_{a < b} p(a)$$

and $<$ reflects symbol ordering. (Thus symbols “a” precede symbol “b” in the ordering, and $P(b)$ is the sum of their probabilities).

Recursive Algorithm on $s$, where $s =$ so-far encoded string, and $b =$ current symbol.

1. Subdivide current interval $C(s)$ and $A(s)$ according to the symbol ordering and respective probabilities.

2. New left endpoint $C(s, b)$ of Interval for symbol “b” is $C(s, b) = C(s) + A(s) \times P(b)$.

3. New Available space (Width) $A(s, b)$ of Interval for symbol “b” is $A(s, b) = A(s) \times p(b)$.

Specialized to binary case (0 and 1)

Define value $Augend(s) = A(s) \times P(1)$. What is added to $C$ for ‘1’.

$Augend(s)$ is the value added to $C(s)$ when symbol 1 is encoded. (When 0 is encoded, $C(s)$ doesn’t change value).

- To Encode 0:
  1. $C(s, 0) = C(s)$. P(0) is 0.
  2. $A(s, 0) = Augend(s)$. Non-zero Augend is $A(s) \times P(1)$.

- To Encode 1:
  1. $C(s, 1) = C(s) + Augend(s)$. P(1) is the Augend.
  2. $A(s, 1) = A(s) - Augend(s)$.

- I.E.: $A(s, 1) = A(s) \times (1 - p(0)) = A(s) \times p(1)$.

Notes on the Recursion Equations for the Binary Case

Note 1: The value $A(s)$ is the product of the probabilities of the so-far encoded data. The ideal length for the code string is thus:

$$-\log_2 \frac{1}{A(s)}$$

Note 2: The values of $p(0)$ and $p(1)$ can actually change from one interval to the next. The decoder need only use the same probability values, as in a higher-order Markov model for example.

Note 3: After encoding a hundred or so binary events, $A(s)$ soon is of the form of a lot of leading 0s: 0.00· · · 01... Also, the number of significant bits behind the first 1 becomes very large, very soon (unless $p(0) = p(1) = \frac{1}{2}$). The large number of significant bits becomes a precision problem in the amount of time and bits involved in doing the multiplication that determines $Augend(s)$ in the recursion.

Algorithmic Version of Graphical Example of Coding C(101)

1. Initial state
   - $C(A) = 0.0$.
   - $A(A) = 1.0$.

2. Encode first bit: 1
   - $Augend(1) = 1 \times 0.2 = 0.2$.
   - $C(1) = 0.0 + 0.2 = 0.2$.
   - $A(1) = 1.0 - Augend(1) = 1.0 - 0.2 = 0.8$.

3. Encode second bit: 0
   - $Augend(0) = 0.8 \times 0.2 = 0.16$.
   - $C(0) = C(1) = 0.2$.
   - $A(0) = Augend(10) = 0.16$.

4. Encode third bit: 1
   - $Augend(101) = 0.16 \times 0.2 = 0.032$.
   - $C(101) = C(10) + 0.032 = 0.2 + 0.032 = 0.232$.
   - $A(101) = A(10) - 0.032 = 0.18$.
Code string termination procedure
For final code point C, pick any point such:

\[ 0.232 \leq C < 0.360 (= 0.232 + 0.128) \]

Pick \( C = 0.3 \): fewest digits.

Algorithmic Method to Decode Code point C
Decoding is by Magnitude Comparisons

**Basic Idea:** For binary case, use one magnitude comparison to test the code string value C to see if it is large enough for the Encoder to have added the Augend.

In 3-ary (or m-ary) alphabets, then two (or m-1) magnitude comparisons are needed to decode the last symbol in the ordering.

In m-ary case decode symbol \( j \) for which

\[ A \times P(j) \leq C < A \times P(j + 1). \]

where, when variable \( j \) takes on value \( m \) when symbols run from 0, \ldots, m-1: so define \( P(m) \) to be 1.0.

**Decoding in Binary Case for Code point C**

**Binary case:** first symbol is 0, second symbol is 1

Form Binary Augend(s): \( \text{Augend}(s) = A(s) \times P(1) \).

- **Decode 0** if \( C(\text{temp}) = (C - \text{Augend}(s)) \) is less than 0.
  
  Note that since \( A \) is 1.0, then the Augend is \( P(1) \) at this point, which is what Encoder would have added to encode 1.
  
  - Recursion for C: do same as Encoder: keep same value of C as before. Note that P(0) is 0, which is what the Encoder would have added to encode 0.
  
  - Recursion for A: do same as Encoder: \( A(\text{next}) = A(\text{current}) \times p(0) \).

- **Decode 1** if \( C(\text{temp}) \) is greater than or equal to 0.
  
  - Recursion for C: New C is C(\text{temp}). Represents subtracting out the Augend value \( (A \times P(1)) \) the encoder had added.
  
  - Recursion for A: do same as Encoder: \( A(\text{next}) = A(\text{current}) \times p(1) \).

Decode C(101) by Algorithmic Method

1. Initial state

\[ \cdot C(A) = 0.3. \]
\[ \cdot A(A) = 1.0. \]

2. Decode first bit: 1

\[ \cdot \text{Augend}(A) = 1 \times 0.2 = 0.2. \]
\[ \cdot C(\text{temp}) = 0.3 - 0.2 = 0.1 \geq 0: \text{Decode 1.} \]
\[ \cdot C(1) = 0.1. \text{ (new } C \text{ is } C(\text{temp}) \]
\[ \cdot A(1) = 1.0 - \text{Augend}(1) = 1.0 - 0.2 = 0.8. \text{ Resize interval.} \]

3. Decode second bit

\[ \cdot \text{Augend}(1) = 0.8 \times 0.2 = 0.16. \]
\[ \cdot C(\text{temp}) = 0.1 - 0.16 = -0.06. \text{ decode 0.} \]
\[ \cdot C(10) = C(1) = 0.1. \]
\[ \cdot A(10) = \text{Augend}(1) = 0.16. \]

4. Decode third bit

\[ \cdot \text{Augend}(10) = 0.16 \times 0.2 = 0.032. \]
\[ \cdot C(\text{temp}) = 0.1 - 0.032 = 0.068. \geq 0. \text{ Decode 1.} \]
\[ \cdot C(101) = C(\text{temp}) = 0.068. \text{ Three symbols decoded: stop decoding.} \]
\[ \cdot A(101) = A(10) - 0.032 = 0.128. \]

**How does decoder know when to stop decoding?**

Technique assumed in example: Header of code string provided the number of symbols to decode, which the decoder stored and counted down to zero.

Another technique is a create an additional symbol in the alphabet called EOF (End of File) that, when decoded, causes the decoder to stop decoding.

**Technical Problem #1: The Precision Problem**

Consider the recursion on \( A(s) \):

\[ A(s, b) = A(s) \times p(b). \]

If \( p(b) \) is to 16 bits of precision, then encoding a file of 64 K bytes, with a 256-ary Elias technique, then the final precision of \( A(s) \) will be: \( 2^{16} \times 2^{16} \). This implies multiplication where the multiplicand has a precision of giga-bits!
Prof Tom Cover of Stanford University suggested to his student Richard Pasco, that he look into practical ways of implementing Elias’ code.

Dr. F. Jelinek of Cornell University had looked into the same problem. Moreover, a coding technique for compression mentioned in Cover’s paper “Enumerative Coding” employed the arithmetic operations of multiplication and addition to form the code string suffered from the same precision problem.

Technical Problem #1: The Precision Problem: Solution

Pasco’s Idea

Let $A(s)$ be 16 bits, let $p(b)$ be 16 bits, and truncate the product $A(s, b)$ to 16 bits. Pasco also created $Augend(s, b) = A(s) \times P(b)$ as a 32-bit product, so he needed to do 32-bit addition.

It turns out that recursion $A(s, b)$ determines the code length, so he could have truncated the Augend as well, as long as the decoder does the same thing.

Earlier, Rissanen had solved the precision problem for enumerative coding, by representing probabilities $p(b)$ by $\ell(b)$, a truncated fraction for $-\log_2 p(b)$. Jorma Rissanen called his result arithmetic coding.

Technical Problem #2: The Carry-over Problem

Defining the Problem

When the sum $C(s, b) = C(s) + A(s) \times P(b) (= Augend(s, b))$, is formed, the trailing bits of $C(s)$ can have a string of 1s:

0111111
If the trailing 1s overlap the leading 1 of $Augend(s, b)$ then the sum is:

100000
Technically, such a carry could propagate over thousands of positions, costing the technique a real-time coding flavor because of unresolved values.

EXAMPLE

| C | 0.011011111111111111111111111111111101110 |
| A | 0.000000000000000000000000000000000110111 |

Pasco took no action, and added the carry as far as it needed to propagate.

Technical Problem #2: The Carry-over Problem

Solutions

The problem of adding the carry over a long chain of 1s was first solved by the encoder algorithm in a paper by Frank Rubin. The solution postpones output, then emits a large burst, so the solution does not retain the real-time property.

Real-time Langdon and Rissanen idea use bit-stuffing.

Follow a certain number of $k$ 1s in the code string $C(s)$ with a stuffed bit whose value is 0. The code string to the left of the stuffed bit position can be transmitted. If a carry occurs, it cannot propagate beyond the stuffed 0.

The decoder, encountering a string of $k$ 1s in the code string, removes the stuffed bit. If value 0, no further action. If value 1, a carry is added into the least significant position above the stuffed position.

EXAMPLE

Let sixteen (16) 1s in succession to left of most significant bit of A cause stuffed position.

To show the stuffed position, use “0” instead of 0 to identify the stuffed bit.


C = 0.011011111111111111111111111111111101110 Augend=0.000000000000000000000000000000000110111
C(new)=0.01101111111111111111111111111111101111

Note that the higher-order bits following the first stuffed “0” did not change. The first string of 16 1s is followed by a “0” so the decoder just throws the “0” away. The second inserted “0” was converted to a 1, so the decoder will propagate the carry.

The carry need not be added to C until the event that created the carry-over is to be decoded. Recall we are dealing in magnitudes, and the cause of the carry-over is the only event affected the position that created the original carry beyond the most significant bit of the A register.

More Carry-over Problem

Retain the least significant byte not all 1 (values 0 – 254) as the “guard byte” to stop the carry. A byte-255 counter is set to 0.

Byte value 255 is all 1s, and will propagate a carry to the next higher byte.

If the next byte and successive are 255, increment the byte-256 counter by 1.

If a carry-over occurs:

1. Propagate the carry to the guard byte.
2. Output the incremented guard byte.

CE 108 Spring 2000

Arithmetic Coding
3. Output the number of bytes of value 0 indicated by
   the byte-256 counter, less one,
4. Reset the byte-256 counter to 0,
5. Retain the lowest-order non-255 byte as the new
   guard byte.

If a new guard byte occurs:

1. Output the former guard byte,
2. Output the number of bytes of value 255 indicated by
   the byte-256 counter,
3. Reset the byte-256 counter to 0,
4. Retain the most recent non-255 byte as the new guard
   byte.

Leading 0s in A register

Fixed registers are used for the coding. In practice, a 32-
bit register can be used for the A and C registers. The A
register value can be represented in 8 hexadecimal digits.
Let unity, or 1.0, be represented in hex as 0x00001000, or
in binary, 0000 0000 0000 0001 0000 0000 0000 0000. This
gives 16 fractional bits. The fractions p(b) and P(b) have
15 fractional bits for their representation in this scheme.

As the A(s) recursion diminishes its value below unity,
the process of renormalization can take place to retain the
precision.

EXAMPLE. Consider the product A(s,b) = A(s) × p(b).
Let A = 1.0 and p(b) = 0.078125 (= 1/16 + 1/64).

\[
\begin{align*}
A &= 0000 0000 0000 0001.0000 0000 0000 0000 \\
p(b) &= 0000 0000 0000 0000.0001 0100 0000 0000 \\
\text{A(s)x}p(b): \\
\text{Prod:} &= 0000 0000 0000 0000.0001 0100 0000 0000 \\
\text{Shift left 4 times. Renormalized A(s,b):} \\
\text{Remr:} &= 0000 0000 0000 0001.0100 0000 0000 0000 
\end{align*}
\]

Comments: Renormalization Procedure

- Each left shift doubles the value of the product.
- The values C(s,b) and A(s,b) must keep their relative
  positions. So whenever A(s,b) is doubled (left-shifted) then so must C(s,b).
- In the case of the A register, we are shifting out
  leading 0s.
- In the case of C(s,b), we are shifting out bits of
  the ultimate code string.
- Define: The higher-order bits are those that occupy
  a position of higher order than the A register
  representation of unity (higher than 0x00010000).
- With a carry-over handling strategy or technique
  implemented, the higher-order bits of C(s,b) can
  be output in real time.