Multimedia Communications

Arithmetic Coding

Arithmetic Coding

- It has been shown that Huffman encoding will generate a code whose rate is within p_{max} +0.086 of the entropy (p_{max} is the probability of the most frequent symbol)
- When the size of the alphabet is small or the probabilities are skewed p_{max} can be quite large
- Huffman codes become inefficient under these conditions
- The performance of Huffman codes can be improved by grouping blocks of symbols together
- This can cause problems in terms of the memory requirements and decoding

Example

- A={a1,a2,a3} p(a1)=0.95, p(a2)=0.02, p(a3)=0.03
- H=0.335 bits/symbol

Letter	Codeword
a1	0
a2	11
a3	10

l=1.05 bits/symbol

- To reduce the redundancy to acceptable level, we should block eight symbols together
- Alphabet size: 6561
- Huge memory,
- Decoding is highly inefficient

Letter	Probability	Codeword
a1a1	0.9025	0
a1a2	0.0190	111
a1a3	0.0285	100
a2a1	0.0190	1101
a2a2	0.0004	110011
a2a3	0.0006	110001
a3a1	0.0285	101
a3a2	0.0006	110010
a3a3	0.0009	110000

l=0.611 bits/symbol

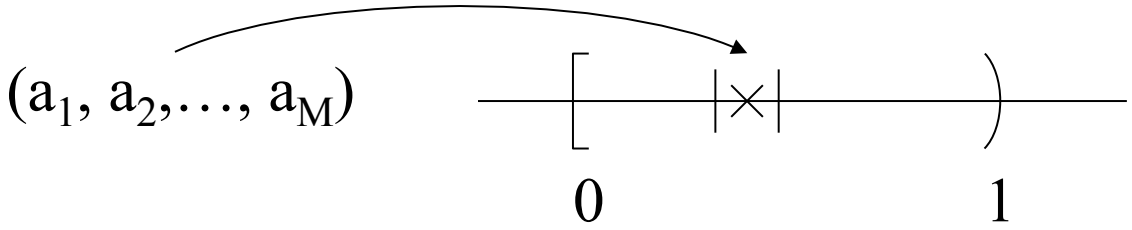
McMaster University

Arithmetic Coding

- It is more efficient to generate codewords for a sequence of symbols
- In Huffman to find codeword for a particular sequence of length M, we have to generate codewords for all possible sequences of length M
- We need a way of assigning codewords to particular sequences without having to generate codes for all sequences of that length
- Arithmetic coding fulfills this requirement

Arithmetic Coding

- Map a sequence into an interval. The sequence can then be encoded by transmitting a tag that represents the interval.



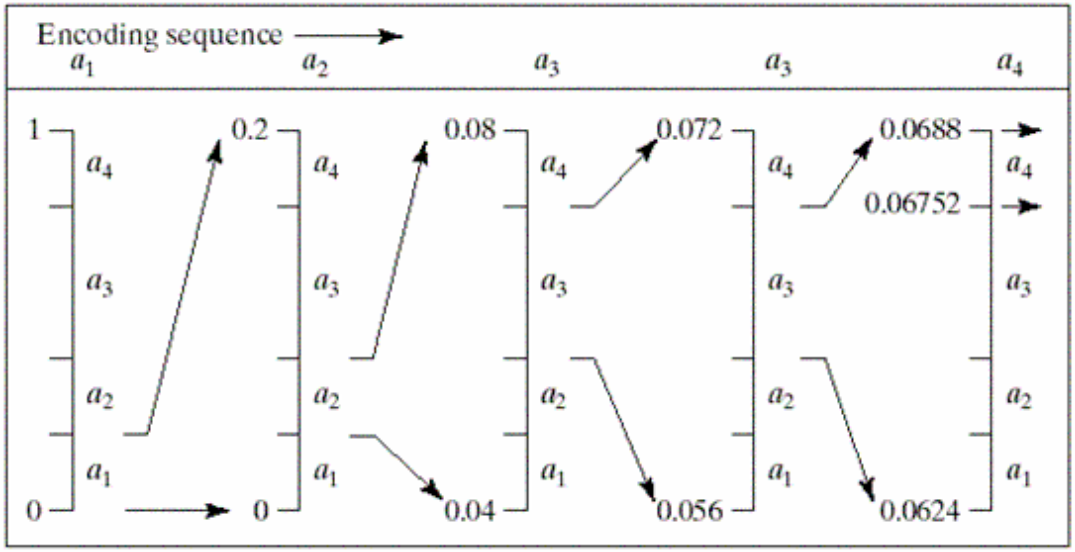
- Key advantage: no need to generate codewords for all sequences of length M.

Arithmetic Coding

- Encoding is constructing and conveying an interval whose length is the product of the probabilities of the input symbols so far encoded.
- Encoding is done recursively. If the sequence $\{a_1,\ldots,a_{n-1}\}$ has been encoded into the interval [l(n-1),u(n-1)), the sequence $\{a_1,\ldots,a_n\}$is encoded into an interval [l(n),u(n)) obtained by
  - subdividing the previous interval into subintervals of lengths proportional to the probabilities of the symbols.
  - choosing the subinterval corresponding to the symbol a_n.

Arithmetic Coding

Source Symbol	Probability	Initial Subinterval
a_1	0.2	[0.0, 0.2)
a_2	0.2	[0.2, 0.4)
a_3	0.4	[0.4, 0.8)
a_4	0.2	[0.8, 1.0)



Arithmetic Coding

- For convenience, we define a random variable X as:

$$X(a_i) = i$$

- If the nth symbol that is encoded is a_i then the lower and upper limits are updated as:

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(i-1)$$

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F_X(i)$$

$$F_X(i) = \sum_{k=1}^{i} P(X = k)$$

Arithmetic Coding: Bit Rate

- What tag to use? Popular choices are:
  - Middle of interval
  - Best: tag requiring smallest number of bits
- If middle of the interval is used as the tag, a binary representation of the tag truncated to

$$l(\underline{x}) = \lceil -\log(p(\underline{x}) \rceil + 1$$

bits is uniquely decodable.

- For this code, we can show that the average length satisfies

$$H(X) \le \bar{l} \le H(X) + \frac{2}{M}$$

Example 1: Encoder

- Encode the sequence "bab".
  For the first symbol, p(a)=2/3 and p(b)=1/3 => $F_x(0)=0$, $F_x(1)=2/3$, $F_x(2)=1$.
  For the second symbol, p(a)=1/2 and p(b)=1/2 => $F_x(0)=0$, $F_x(1)=1/2$, $F_x(2)=1$.
  For the third symbol, p(a)=2/5 and p(b)=3/5 => $F_x(0)=0$, $F_x(1)=2/5$, $F_x(2)=1$.
- Set l(0) = 0 and u(0) = 1.
- Input "b": l(1) = 0 + (1) (0.66)=0.66 and u(1) = (1) (1) = 1
- Input "a": l(2) = 0.66+(0.33)(0) = 0.66 and u(2) = 0.66+(0.33) (0.50) =0.83.

Example 1: Encoder

- Input "b": $l(3) = 0.66 + (0.16)\ (0.40) = 0.73$ and $u(3) = 0.66+(0.16)(1) = 0.83$.
- Since $[0.110000..., 0.110100...) \subset [0.73, 0.83)$, send **1100**, which is the shortest uniquely decodable code.
- Unique decodability implies that whatever bits the decoder adds to the code, the resulting real number still lies within the current interval.

Example 1: Encoder

- For example, if we only send **110** and the decoder adds **1** bits to the right side of the code, the resulting real number will definitely be larger than 0.83. However, sending **1100** will leave no ambiguity. In fact, you can check that $0.11001111 \in [0.73, 0.83)$.

Example 1: Decoder

- Decode the sequence **1100**. Construct C = 0.11001000… (0.78125), where the right four bits are random bits. Assume we have a copy of the model used at the encoder.
- Set l(0)=0 and u(0)=1.
- Subdivide [0,1) to [0,0.66) and [0.66,1).
- Since C=0.78125 ∈ [0.66,1), select [0.66,1) and emit "b".
- Subdivide [0.66,1) to [0.66, 0.83) and [0.83,1).
- Since C ∈ [0.66, 0.83), emit "a".
- Subdivide [0.66, 0.83) to [0.66, 0.73) and [0.73,0.83).
- Since C ∈ [0.73,0.83), emit "b". End.

Advantages and Problems

- Advantages:
  - Naturally suitable for adaptation strategies.
  - Close-to-optimal compression performance for sources with very low entropies.
- Problems:
  - Unlike Huffman coding, the decoder requires (explicit or implicit) knowledge of the number of symbols to be decoded.
  - Transmission of coded data cannot begin until the whole sequence is encoded.
  - Infinite precision is required.

- There are two way to know when the entire sequence has been decoded:

1. The decoder may know the length of the sequence in which case the process is stopped when that many symbols have been obtained
2. A particular symbol is denoted as an end-of-transmission symbol. The encoding of this symbol would bring the decoding process to a close.

Incremental coding

- As the number of symbols coded, gets larger, the values of l(n) and u(n) get closer to each other
- This means that in order to represent all subintervals we need increasing precision
- We would also like to perform encoding incrementally: transmit portions of the code as the sequence is being observed

Incremental coding

- Once the interval is confined to either upper or lower half of the unit interval, it is forever confined to that interval
- The most significant bit of the binary representation of all numbers in the interval [0,0.5) is 0
- The most significant bit of the binary representation of all numbers in the interval [0.5,1) is 1
- Therefore, without waiting to see that the rest of the sequence looks like, we can indicate to the decoder whether the tag is confined to the upper or lower half of the unit interval by sending 1 for upper half and 0 for the lower half

Incremental coding

- Once the encoder and decoder know which half contains the tag, we can ignore the other half and map the half containing the tag to the full [0,1) interval.
  - E1: [0,0.5) -> [0,1) E1(x)=2x
  - E2: [0.5,1) -> [0,1) E2(x)=2(x-0.5)
- A method to prevent the current interval from becoming too small when the interval is short but includes 0.5.
- Suppose [l,u) satisfies: $0.25 \leq l < 0.5 < u \leq 0.75$
- This is called underflow conditions.
- The next bit to output is still unknown, but the next two bits should be 01 or 10
- Why? if the next bit is 0 after that we cannot have another 0 because we will fall out of [0.25,0.75]. Similar reasoning for 1.

Incremental coding

- What kind of expansion under this condition?
- Expand [l,u) about 0.5 by E3(x)=2(x-0.25)
- Let's study effects of this expansion:
  - If r=(.01a3a4…) belongs to [l,u) then expansion will give (.0a3a4…)
  - If r=(.10a3a4…) belongs to [l,u) then expansion will give (.1a3a4…)
- Underflow expansion preserves the code stream
- After expansion the first bit is the same as the first bit before expansion

Incremental coding

- Algorithm for arithmetic coding with rescaling:
  1. Current interval [l,u) is initialized to [0,1). Underflow count is initialized at 0.
  2. If $0.25 \le l < 0.5 < u \le 0.75$ expand to [2(l-0.25),2(u-0.25)) and increment the underflow count
  3. If $[l,u) \subset [0,0.5)$ then output 0 and any pending underflow bits, all 1, and expand the current interval to [2l,2u). If $[l,u) \subset [0.5,1)$ then output 1 and any pending underflow bits, all 0, and expand the current interval to [2(l-0.5),2(u-0.5)). In either case, reset the underflow count to zero
  4. If 2 or 3 does not hold, divide the current interval into disjoint subintervals and take the one corresponding to the next symbol

Incremental coding

5. Repeat steps 2-4 until there are no more source letters and none of the conditions in 2 or 3 hold. At this stage, the final interval satisfies $l<0.25<0.5<u$ or $l<0.5<0.75<u$. Any point inside the interval could be transmitted. We pick 0.25 (01 in binary) for the first and 0.5 (10) for the second one. Any underflow bits are output after the first of these bits.

Integer Implementation of Arithmetic Coding

- $l^{(0)}$=00…0 (m bits) and $u^{(0)}$=11…1 (m bits)
- n_j: number of times the symbol j occurs in a sequence of length Total_Count

$$Cum\_Count(k) = \sum_{i=1}^{k} n_i$$

$$l^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_Count(k-1)}{Total\_Count} \right\rfloor$$

$$u^{(n)} = l^{(n-1)} + \left\lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1) \times Cum\_Count(k)}{Total\_Count} \right\rfloor - 1$$

Integer Implementation of Arithmetic Coding (encoder)

- If MSB of both $u^{(n)}$ and $l^{(n)}$ are zero or one, shift it out and shift in a 1 into the LSB of $u^{(n)}$ and a zero into LSB of $l^{(n)}$. Send as many as underflow counter of the complement of the bit shifted out.
- If the second MSB of $u^{(n)}$ is 0 and the second MSB of $l^{(n)}$ is 1, complement the second MSB of $u^{(n)}$ and $l^{(n)}$, shift left, shifting in a 1 in $u^{(n)}$ and a 0 into $l^{(n)}$. Increment the underflow counter.
- Continue until the last symbol. After coding the last symbol sent the lower limit as the tag. If underflow counter is not zero after sending the first bit of lower limit send as many as the counter of the opposite bit and then the remaining bits of lower limit.

Integer Implementation of Arithmetic Coding (decoder)

- If MSB of both $u^{(n)}$ and $l^{(n)}$ are zero or one, shift it out and shift in a 1 into the LSB of $u^{(n)}$ and a zero into LSB of $l^{(n)}$. Shift tag to the left one bit and read the next bit from the received bitstream
- If the second MSB of $u^{(n)}$ is 0 and the second MSB of $l^{(n)}$ is 1, complement the second MSB of $u^{(n)}$ and $l^{(n)}$, shift left, shifting in a 1 in $u^{(n)}$ and a 0 into $l^{(n)}$. Complement the second MSB of the tag, shift the tag to the left one bit read the next bit from the received bitstream

Comparison

- For arithmetic coding:

$$H(X) \le \bar{l} \le H(X) + \frac{2}{M}$$

- For Huffman coding

$$H(S) \le \bar{l} \le H(S) + \frac{1}{M}$$

- On the surface it seems that the advantage is with Huffman
- It is impractical to build long enough sequences with Huffman
- It is feasible to build long sequences for arithmetic coder