CSC310 – Information Theory — Sam Roweis

Lecture 8:

Arithmetic Coding – Details

October 5, 2005

Stream Codes: Transmitting Bits as We Go 1

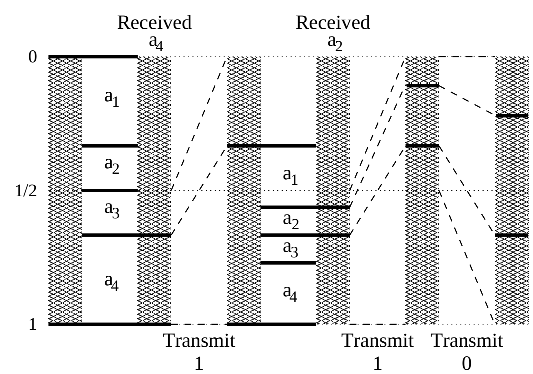
- The problem of needing high-precision arithmetic makes aritmetic coding potentially impractical. We'll try to solve it by transmitting bits as soon as they are determined.
- **Example:** After looking at the first few symbols in our block, our interval has been reduced to $[0.625,\, 0.875) \;=\; [0.101_2,\, 0.111_2)$.
- *Any* number in this interval that we might eventually transmit will start with a 1 bit. So we can transmit this bit immediately, without even looking at what symbols come next!
- This scheme is called *stream coding* because we just receive an incoming stream of source symbols and output bits of the encoding as we compute them.
- There is no need for an explicit block length anymore! We are in effect transmitting the entire message as a single block by specifying (to adequate precision) the corresponding interval in $[0, 1)$.

Expanding the Interval After Transmitting a Bit 2

- Once we transmit a bit that is determined by the current interval, we can throw that bit away, and then expand the interval by moving the "bit point" one place to the right and doubling.
- **Example:** Continuing from the previous slide, the interval $[0.625,\, 0.875) \;=\; [0.101_2,\, 0.111_2)$ results in transmission of a 1. We then throw out the 1, and double the bounds, giving the interval $[0.010_2,\, 0.110_2)$.
- In fact, as soon as the interval shrinks to a width of *less than one half*, we will transmit a bit and then double the interval size.
- Hopefully, expanding the interval will allow us to use numerical representations of the bounds, u and v, that are of lower precision.

Picture Of How it Works 3

Suppose we are encoding symbols from the alphabet $\{a_1,\, a_2,\, a_3,\, a_4\}$, with probabilities $1/3,\, 1/6,\, 1/6,\, 1/3$.
Here's how the interval changes as we encode the message $a_4,\, a_2,\, \ldots$



Arithmetic Coding Without Blocks (ver 1.0) 4

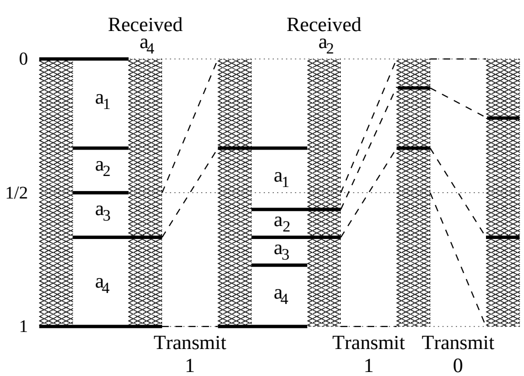
1) Initialize interval $[u, v)$ to $u = 0$ and $v = 1$.

2) For each source symbol, a_i, in turn:
   Compute $r = v - u$.
   Let $u = u + r \sum_{j=1}^{i-1} p_j$. Let $v = u + rp_i$.
   While $u \geq 1/2$ or $v \leq 1/2$:
     If $u \geq 1/2$:
       Transmit a 1 bit.
       Let $u = 2(u-1/2)$ and $v = 2(v-1/2)$.
     If $v \leq 1/2$:
       Transmit a 0 bit.
       Let $u = 2u$ and $v = 2v$.

3) Transmit enough final bits to specify a number in $[u, v)$.

But wait... 5

Suppose we are encoding symbols from the alphabet $\{a_1, a_2, a_3, a_4\}$, with probabilities $1/3, 1/6, 1/6, 1/3$.
Here's how the interval changes as we encode the message $a_4, a_2, \ldots$



Now it looks like we are stuck. The interval corresponding to a_2 is still less than one half in width (4/9), but we can't transmit another bit yet.

Precision Might Still be a Problem... 6

- We hope that by transmitting bits early and expanding the interval, we can avoid tiny intervals, requiring high precision to represent.
- *Problem:* What if the interval gets smaller and smaller, *but it always includes 1/2?*
- For example, as we encode symbols, we might get intervals of:
  $[0.00000_2,\ 1.00000_2)$
  $[0.01010_2,\ 0.11001_2)$
  $[0.01101_2,\ 0.10100_2)$
  $[0.01111_2,\ 0.10010_2)$
  $\ldots$
- Although the interval is getting smaller and smaller, we still can't tell whether the next bit to transmit is a 0 or a 1.

A Solution 7

- When a narrow interval straddles 1/2, it will have the form
$$[0.01xxx,\ 0.10xxx)$$
- So although we don't know what the next it to transmit is, we *do* know that the bit transmitted after the next will be the opposite.
- We can therefore expand the interval around the *middle* of the range, remembering that the next bit output should be followed by an opposite bit.
- If we need to do several such expansions, there will be several opposite bits to output, but we will remember them all.
- We have to be a bit less ambitious, and transmit a bit only when the interval size shrinks below *one quarter*, but that's still OK in terms of precision.

Arithmetic Coding Without Blocks (ver 1.1)

1) Initialize the interval $[u,\ v)$ to $u = 0$ and $v = 1$.
   Initialize the "opposite bit count" to $c = 0$.

2) For each source symbol, a_i, in turn:
   Compute $r = v - u$.
   Let $u = u + r \sum_{j=1}^{i-1} p_j$. Let $v = u + rp_i$.
   While $u \geq 1/2$ or $v \leq 1/2$ or $u \geq 1/4$ and $v \leq 3/4$:
   - If $u \geq 1/2$:
     - Transmit a 1 bit followed by c 0 bits. Set c to 0.
     - Let $u = 2(u-1/2)$ and $v = 2(v-1/2)$.
   - If $v \leq 1/2$:
     - Transmit a 0 bit followed by c 1 bits. Set c to 0.
     - Let $u = 2u$ and $v = 2v$.
   - If $u \geq 1/4$ and $v \leq 3/4$:
     - Set c to $c + 1$.
     - Let $u = 2(u-1/4)$ and $v = 2(v-1/4)$.

3) Transmit enough final bits to specify a number in $[u,\ v)$.

What Have We Gained?

- By expanding the interval in this way, we ensure that the size of the (expanded) interval, $v - u$, will always be at least 1/4.
- We can now represent u and v with a *fixed* amount of precision — we *don't* need more precision for longer messages.
- We will use a *fixed point* (scaled integer) representation for u & v.
- Why not floating point?
  - Fixed point arithmetic is faster on most machines.
  - Fixed point arithmetic is well defined. Floating point arithmetic may vary slightly from machine to machine. The effect? Machine B might not correctly decode a file encoded on Machine A!

Quantized Symbol Probabilities

- To completely do away with floating point operations, we need to represent the symbol probabilities as rational fractions with large denominators (to give sufficient precision).
- We'll estimated the probabilities of symbols $a_1, \ldots, a_i$ using the fractions

  $$p_i \ = \ f_i \,/\, \sum_{j=1}^{I} f_j$$

  (with all $f_i > 0$).
  For arithmetic coding, it's convenient to pre-compute the *cumulative frequencies*

  $$F_i \ = \ \sum_{j=1}^{i} f_j$$

  We define $F_0 = 0$, and use T for the total (denominator) F_I. We will assume that $T < 2^h$, so the f_i take no more than h bits.

Precision of the Coding Interval

- The ends of the coding interval will be represented by m-bit integers. The integer bounds u and v represent the interval

  $$\big[\, u \times 2^{-m},\ (v{+}1) \times 2^{-m} \big)$$

  (The addition of 1 to v allows the upper bound to be 1 without the need to use $m{+}1$ bits to represent v.)
- The received message will be represented as an m-bit integer, t, plus further bits not yet read.
- With these representations, the arithmetic performed will never produce a result bigger than $m + h$ bits.

Encoding Using Integer Arithmetic

$u \leftarrow 0,\ v \leftarrow 2^m - 1,\ c \leftarrow 0$
For each source symbol, a_i, in turn:
  $r \leftarrow v - u + 1$
  $v \leftarrow u + \lfloor (r * F_i) \,/\, T \rfloor - 1$
  $u \leftarrow u + \lfloor (r * F_{i-1}) \,/\, T \rfloor$
  While $u \geq 2^m/2$ or $v < 2^m/2$ or $u \geq 2^m/4$ and $v < 2^m * 3/4$:
    If $u \geq 2^m/2$:
      Transmit a 1 bit followed by c 0 bits
      $c \leftarrow 0$
      $u \leftarrow 2 * (u - 2^m/2),\ v \leftarrow 2 * (v - 2^m/2) + 1$
    If $v < 2^m/2$:
      Transmit a 0 bit followed by c 1 bits
      $c \leftarrow 0$
      $u \leftarrow 2 * u,\ v \leftarrow 2 * v + 1$
    If $u \geq 2^m/4$ and $v < 2^m * 3/4$:
      $c \leftarrow c + 1$
      $u \leftarrow 2 * (u - 2^m/4),\ v \leftarrow 2 * (v - 2^m/4) + 1$

Transmit a few final bits to specify a point in the interval
If $u < 2^m/4$: Transmit a 0 bit followed by c 1 bits. Then transmit a 1 bit
Else: Transmit a 1 bit followed by c 0 bits. Then transmit a 0 bit.

Precision Required

- For this procedure to work properly, the loop that expands the interval must terminate. This requires that the interval never shrink to nothing — ie, we must always have $v \geq u$.
- This will be guaranteed as long as

  $$\lfloor (r * F_i) \,/\, T \rfloor \ > \ \lfloor (r * F_{i-1}) \,/\, T \rfloor$$

  This will be so as long as $f_i \geq 1$ (and hence $F_i \geq F_{i-1} + 1$) and $r \geq T$.
- The expansion of the interval guarantees that $r \geq 2^m/4 + 1$.
- So the procedure will work as long as $T \leq 2^m/4 + 1$. If our symbol counts are bigger than this, we have to scale them down (or use more precise arithmetic, with a bigger m).
- However, to obtain near-optimal coding, T should be a fair amount less than $2^m/4 + 1$.

Decoding Using Integer Arithmetic

$u \leftarrow 0,\ v \leftarrow 2^m - 1$
$t \leftarrow$ first m bits of the received message
Until last symbol decoded:
  $r \leftarrow v - u + 1$
  $w \leftarrow \lfloor ((t - u + 1) * T - 1) \,/\, r \rfloor$
  Find i such that $F_{i-1} \leq w < F_i$
  Output a_i as the next decoded symbol
  $v \leftarrow u + \lfloor (r * F_i) \,/\, T \rfloor - 1$
  $u \leftarrow u + \lfloor (r * F_{i-1}) \,/\, T \rfloor$
  While $u \geq 2^m/2$ or $v < 2^m/2$ or $u \geq 2^m/4$ and $v < 2^m * 3/4$:
    If $u \geq 2^m/2$:
      $u \leftarrow 2 * (u - 2^m/2),\ v \leftarrow 2 * (v - 2^m/2) + 1$
      $t \leftarrow 2 * (t - 2^m/2)\ +$ next message bit
    If $v < 2^m/2$:
      $u \leftarrow 2 * u,\ v \leftarrow 2 * v + 1$
      $t \leftarrow 2 * t\ +$ next message bit
    If $u \geq 2^m/4$ and $v < 2^m * 3/4$:
      $u \leftarrow 2 * (u - 2^m/4),\ v \leftarrow 2 * (v - 2^m/4) + 1$
      $t \leftarrow 2 * (t - 2^m/4)\ +$ next message bit

Proving That the Decoder Finds the Right Symbol

- To show this, we need to show that if

  $$F_{i-1} \ \leq\ \lfloor ((t - u + 1) * T - 1) \,/\, r \rfloor \ <\ F_i$$

  then

  $$u + \lfloor (r * F_{i-1}) \,/\, T \rfloor \ \leq\ t \ \leq\ u + \lfloor (r * F_i) \,/\, T \rfloor - 1$$

- This can be proved as follows:

$$F_{i-1} \ \leq\ \lfloor ((t - u + 1) * T - 1) \,/\, r \rfloor \ \leq\ ((t - u + 1) * T - 1) \,/\, r$$
$$\Rightarrow\ r * F_{i-1} \,/\, T \ \leq\ t - u + 1 - 1/T$$
$$\Rightarrow\ u + \lfloor (r * F_{i-1}) \,/\, T \rfloor \ \leq\ u + (t - u) \ =\ t$$

$$F_i \ >\ \lfloor ((t - u + 1) * T - 1) \,/\, r \rfloor$$
$$\Rightarrow\ F_i \ \geq\ \lfloor ((t - u + 1) * T - 1) \,/\, r \rfloor \ +\ 1$$
$$\Rightarrow\ F_i \ \geq\ ((t - u + 1) * T - 1) \,/\, r \ -\ (r - 1)/r \ +\ 1$$
$$\Rightarrow\ r * F_i \,/\, T \ \geq\ t - u + 1 - 1/T \ -\ (r - 1)/T \ +\ r/T$$
$$\Rightarrow\ r * F_i \,/\, T \ \geq\ t - u + 1$$
$$\Rightarrow\ u + \lfloor (r * F_i) \,/\, T \rfloor - 1 \ \geq\ t$$

Summary

- Arithmetic coding provides a practical way of encoding a source in a very nearly optimal way.
- Faster arithmetic coding methods that avoid multiplies and divides have been devised.
- **However:** It's not necessarily the best solution to *every* problem. Sometimes Huffman coding is faster and almost as good. Other codes may also be useful.
- Arithmetic coding is particularly useful for *adaptive* codes, in which probabilities constantly change. We just update the table of cumulative frequencies as we go.

History of Arithmetic Coding

- Elias — around 1960.
  Seen as a mathematical curiosity.
- Pasco, Rissanen – 1976.
  The beginnings of practicality.
- Rissanen, Langdon, Rubin, Jones – 1979.
  Fully practical methods.
- Langdon, Witten/Neal/Cleary — 1980's.
  Popularization.
- Many more... (eg, Moffat/Neal/Witten)
  Further refinements to the method.