COM3110/4115/6115: Text Processing

Text Compression: Arithmetic coding

Rob Gaizauskas

Department of Computer Science
University of Sheffield

Overview

- Models
  - Static
  - Semi-static
  - Adaptive
- Coding
  - Huffman Coding
  - Arithmetic Coding
- Further topics:
  - Symbolwise Models
  - Dictionary Methods
  - Synchronisation
  - Performance Issues

Arithmetic Coding

- Arithmetic coding allows excellent compression
  - ◇ can code arbitrarily close to the entropy
  - ◇ hence is *optimal* terms of compression
- Wins over Huffman coding if distribution is *very skewed*

  e.g. for two letter alphabet $\{a, b\}$ where $Pr[a] = .99$ and $Pr[b] = 0.01$
  - a can be coded in $-log_2 Pr[s] = 0.015$ bits
  - Huffman coding, however, requires at least one-bit/symbol
  - same not true for arithmetic coding
- Arithmetic coding suitable for sophisticated adaptive models
- Disadvantages:
  - ◇ slower than Huffman coding
  - ◇ not easy to start decoding in middle of compressed stream
  - ◇ hence less suitable for full-text retrieval (where random access to compressed text may be needed)
- *Thus*: Huffman most useful for text; arithmetic coding for images

Arithmetic Coding (ctd)

- Output of arithmetic coder is a stream of bits, but think of it as a fractional binary number between 0 and 1:

e.g.

$$0.1011001$$

$$\tfrac{1}{2} \quad \tfrac{1}{4} \quad \tfrac{1}{8} \quad \tfrac{1}{16} \quad \tfrac{1}{32} \quad \tfrac{1}{64} \quad \tfrac{1}{128}$$

$$\frac{1}{2} + \frac{1}{8} + \frac{1}{16} + \frac{1}{128} = \sim 0.695$$

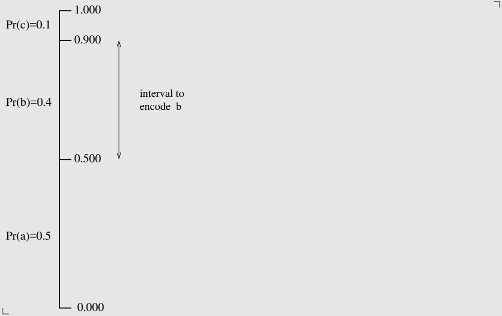
  - ◇ can drop the `"0."` prefix, as same on all outputs

- Suppose we have a ternary alphabet $\{a, b, c\}$, and want to compress text *bab*
  - ◇ assume (static) model: Pr(a)=0.5 Pr(b)=0.4 Pr(c)=0.1

- Arithmetic coder stores two numbers – *high* and *low* – representing a subinterval of $[0, 1]$ used to code next symbol
  - ◇ initially $high = 1$ and $low = 0$

Arithmetic Coding (ctd)

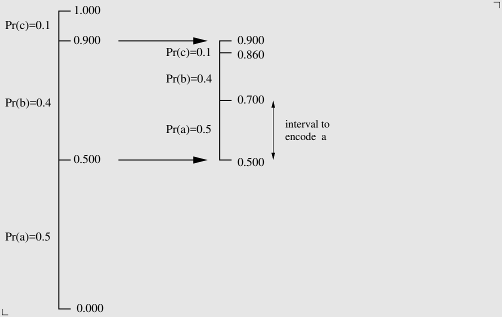
- Range between *high* and *low* sub-divided according to probability distribution of model: sub-intervals allocated for coding each symbol



- The coding step involves resetting the *high*/*low* values to *narrow* the recorded interval
  - here, to code b, set $high = 0.9$ and $low = 0.5$

Arithmetic Coding (ctd)

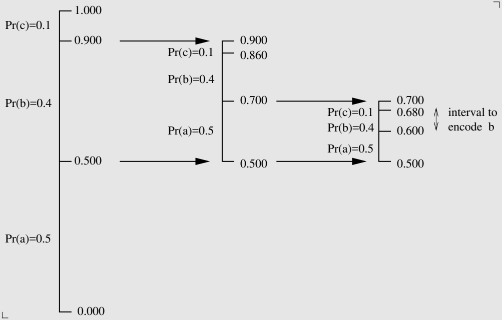
- Coding continues by sub-dividing new current interval (between *high*/*low*) and narrowing to sub-interval for coded symbol



  - ◇ here, set $high = 0.7$ and $low = 0.5$, to code symbol a

Arithmetic Coding (ctd)

- Repeat process to code third symbol *b*:



◇ set *high* $= 0.68$ and *low* $= 0.6$, to code final symbol *b*

◇ final interval represents full input *bab*

Arithmetic Coding (ctd)

- Encoder has processed full input *bab*
  - coding process has narrowed interval so: $high = 0.68$, $low = 0.6$
- Encoder now transmits this content by outputting *any number* in the range between *high* and *low*
  - choose number with *shortest code* (of course!)

  e.g. choose `101` (0.625), not `10101` ($\sim$0.656), though both in range
- Transmitted code, plus model, is sufficient for decoding
  - decoder simulates steps in encoding process
  - and as it does, recovers encoded content
- The smaller the final interval is, the more bits that will be needed to specify a number that falls within it

  e.g. for our example, low probability input *ccc* has final interval width 0.001, and requires more bits to transmit, c.f. input *aaa* with interval width 0.125

Arithmetic Coding (ctd)

- Intuitively, method works because high probability events narrow the interval much less than low probability events do
- The number of bits required is proportional to the negative logarithm of the size of the interval
  - ◇ the interval size is the product of the probabilities of the coded symbols
  - ◇ the logarithm of this quantity is the sum of the logarithms of the individual probabilities
  - ◇ symbol s of probability $P[s]$ contributes $-log_2 P[s]$ bits to output
    - this is equivalent to the symbol's information content
- Hence method is near-optimal
  - ◇ code size identical to the theoretical bound given by the entropy
  - ◇ high probability symbols can be coded in a fraction of a bit

Arithmetic Coding (ctd)

- Method is limited in practice by:
  - ◇ need to transmit, eventually, a whole number of bits/bytes
  - ◇ limited precision arithmetic
- As described, method produces no output until encoding complete

  i.e. until all input processed
- In practice, possible to output bits *during* coding
  - ◇ avoids having to work with higher and higher precision numbers
  - ◇ key observation: when range is *small*, *high*/*low* have *common prefix*

  e.g. if range is $high = 0.6667$, $low = 0.6334$ might:
    - output bits for prefix (here 6)
    - reset range as $high = 0.667$, $low = 0.334$
  - ◇ allows output to be generated *incrementally*

Arithmetic Coding (ctd)

- Arithmetic coding more commonly used with *adaptive modelling*
  - probabilities used based on counts observed in text
- Consider earlier example with alphabet $\{a, b, c\}$, coding input *bab*:
  - initialiase counts to 1 (avoid zero-frequency problem)
  - initial probabilities then: $Pr(a) = \frac{1}{3}$ $Pr(b) = \frac{1}{3}$ $Pr(c) = \frac{1}{3}$
  - this model used to code first character b — then counts updated
  - updated model then: $Pr(a) = \frac{1}{4}$ $Pr(b) = \frac{2}{4}$ $Pr(c) = \frac{1}{4}$
  - after 2nd char a coded, model: $Pr(a) = \frac{2}{5}$ $Pr(b) = \frac{2}{5}$ $Pr(c) = \frac{1}{5}$
  - after 3nd char b coded, model: $Pr(a) = \frac{2}{6}$ $Pr(b) = \frac{3}{6}$ $Pr(c) = \frac{1}{6}$
  - and so on ...
  - see Witten *et al.* for a worked example
- As before, given output, decoder can simulate the encoding process, including all the counting/model update

Reading

- I. H. Witten, A. Moffat, T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd ed. Morgan Kaufmann. 1999.
- Baeza-Yates and Ribeiro-Neto, Modern Information Retrieval, Addison Wesley
- Nam Phamdo. Theory of Data Compression. `www.data-compression.com/theory.html`