COM3110/4115/6115: Text Processing

Text Compression: dictionary methods + further topics

Rob Gaizauskas

Department of Computer Science
University of Sheffield

Overview

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

Dictionary Methods

- Dictionary methods rely on replacing substrings in a text with codewords, or indices, that identify the substring in a *dictionary* or *codebook*
- Simple example: *digram coding* — simple method for ASCII text
  - ◇ uses an 8-bit code dictionary where
    - 128 ASCII characters represent themselves
    - 128 codes represent common letter pairs
  - ◇ at worst each 7-bit character is expanded to 8-bits
  - ◇ at best character pairs (14 bits) are reduced to 8 bits
- Such an approach can be extended to include larger dictionary entries

  e.g. common words such as *and* and *the*

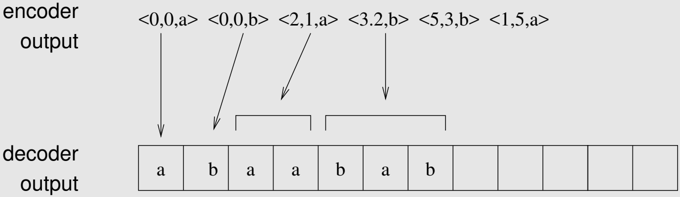
  e.g. common prefixes/suffixes, such as *pre*, *tion*

Dictionary Methods (ctd)

- However, difficult to get good compression with a general dictionary
  - ◇ words common to many texts tend to be short
  - ◇ dictionary suited to one sort of text, poor for another
- Can move to a semi-static scheme where a new codebook is constructed for each text. *But* some obvious drawbacks:
  - ◇ inefficiencies in transmitting dictionary
  - ◇ hard to decide which words to select for dictionary
- Solution: use an adaptive dictionary scheme
  - ◇ most such schemes based on two methods proposed by Ziv and Lempel in 1977/78 (known as LZ77 and LZ78)
  - ◇ key idea: replace substring with a pointer to previous occurrence in same text

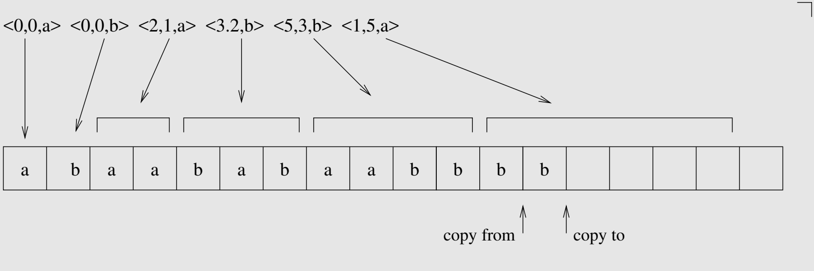
Dictionary Methods: LZ77

- LZ77 can be explained most easily in terms of an example of its decoding (example from Witten *et al.*)
- Suppose alphabet is just $\{a, b\}$



- Encoder output is a series of triples
  - ◇ first component indicates how far back in decoded output to look for next phrase
  - ◇ second indicates the length of that phrase
  - ◇ third is next character from input (only necessary when not found in previous text, but included for simplicity)

Dictionary Methods: LZ77 (ctd)



- Next, decode characters represented by triple `<5,3,b>`
  - go back 5 characters (to 3rd from start)
  - copy 3 characters (*aab*)
  - add the 3rd item from triple – *b* – to this
- Next triple, `<1,5,a>`, is a 'recursive reference'
  - go back one character (*b*)
  - *sequentially* copy the next 5 characters

    — when 2nd char needed, it's available (as a copy of first)
  - results in the addition of 5 consecutive *b*'s and an *a*

Dictionary Methods: LZ77 (ctd)

- Various issues must be addressed in implementing an adaptive dictionary method such as LZ77, including . . .
- How far back in the text to allow pointers to refer
  - ◇ references further back increase chance of longer matching strings, but also increase bits required to store pointer
  - ◇ typical value is a few thousand characters
- How large the strings referred to can be
  - ◇ again, the larger the string, the larger the width parameter specifying it
  - ◇ typical value $\sim$ 16 characters
- During encoding, how to search window of prior text for longest match with the upcoming phrase
  - ◇ linear search very inefficient
  - ◇ best to index prior text with a suitable data structure, such as a trie, hash, or binary search tree

Dictionary Methods: *gzip*

- A popular high performance implementation of LZ77 is *gzip*
- Uses a hash table to locate previous occurrences of strings
  - ◇ hash accessed by next 3 characters
  - ◇ holds pointers to prior locations of the 3 characters
- Pointers and phrase lengths stored using variable length Huffman codes
  - ◇ computed semi-statically by processing 64K blocks of data at a time
  - ◇ this much can easily be held in memory, so appears as if single-pass
- Pointer triples are reduced to pairs, by eliminating 3rd element
  - ◇ common Huffman coding used to transmit *both* phrase lengths *and* characters
    - this is transmitted first
  - ◇ if a phrase length was tranmitted, then next unit will be a pointer

Symbolwise Models

- As we have seen, symbolwise methods work by estimating the probabilities of symbols (characters/words) and coding one symbol at a time using shorter codewords for the more likely symbols
- So far, only *specific symbolwise model* considered (in examples, etc) has been zero order character models

  i.e. character models that do not consider left context

  - ◇ not a serious contender for effective text compression
- PPM is a character-based symbolwise model that uses left context
  - ◇ PPM = *prediction by partial matching*
  - ◇ determines probabilities in a given (finite) context
  - ◇ *but* length of context may vary depending on contexts previously seen
  - ◇ obtains arguably the best compression performance

Symbolwise Models: Word-based Models

- So far have assumed "symbols" are characters
- In word-based models, symbols are "words" (alphanumeric strings) and "non-words" (white-space and punctuation)
- Documents are assumed to consist of strictly alternating words and non-words and (typically)
  - one zero-order model is built for words
  - another zero-order model is built for nonwords
- If an adaptive model is used, a mechanism is needed for previously unseen words/nonwords

  e.g. send an escape symbol then spell out the word character by character, using a zero-order model of characters

Symbolwise Models: Word-based Models (ctd)

- "Parsing" text into words-nonwords raises a number of issues
  - ◇ how is punctuation that is part of a word (hyphens, apostrophes) to be handled?
  - ◇ should every numeric string be a separate word?
    - can lead to huge numbers of words that occur once only
  - ◇ what about ideographic languages?
    - segmenting text into words much more challenging for these
- Word-based models can yield large numbers of symbols
  - ◇ efficient data structure for model important
    - canonical Huffman code good for static/semi-static versions
- Can achieve compression performance close to PPM
  - ◇ with (semi-) static models supports random access

Synchronisation

- Good compression techniques work best on large files
  - ◇ decompression techniques inherently sequential
  - ◇ tends to preclude random access
  - ◇ full-text retrieval systems *require random access*
    - need to consider special measures to facilitate random access for these applications
- Good compression methods make random access difficult, because
  - ◇ use variable length codes
    - can't start decoding at random point, since may not be on codeword boundary
  - ◇ use adaptive models
    - cannot determine model without decoding all prior text
- No good solution for adaptive modelling
- Best to use static models for full text retrieval

Synchronisation (ctd)

- Techniques have been developed for achieving random access in compressed files
- **Synchronisation points**
  - assume smallest unit of random access in compressed archive is the *document*
  - *either* store bit offset of document
  - *or* ensure it ends on byte boundary
- **Self-synchronising codes**
  - design code so that regardless of where decoding starts, comes into synchronisation rapidly and stays there
  - problematic for full text retrieval
    - not possible to guarantee how quickly synchronisation will be achieved

Performance Issues

- Performance considerations for compression algorithms include
  - speed
  - memory
  - compression rate (% remaining or % removed)
- Some methods to compare for performance (from Witten *et al.*):
  - tests were performed on a benchmark corpus that included: a novel, fax bitmap, C source code, Excel spreadsheet, executable code, etc.

Method	Description
pack	zero-order, character-based, semi-static Huffman coder
char	zero-order, character-based, adaptive arithmetic coder
ppm	variable-order, character-based, adaptive arithmetic coder
huffword	zero-order, word-based, semi-static Huffman coder
gzip-f	LZ77-type, semi-static coder, fast option
gzip-b	LZ77-type, semi-static coder, best compression
compress	LZ78-type, Unix *compress* utility
null	copy input to output (Unix *cat*)

Performance Issues (ctd)

Method	Description
pack	zero-order, character-based, semi-static Huffman coder
char	zero-order, character-based, adaptive arithmetic coder
ppm	variable-order, character-based, adaptive arithmetic coder
huffword	zero-order, word-based, semi-static Huffman coder
gzip-f	LZ77-type, semi-static coder, fast option
gzip-b	LZ77-type, semi-static coder, best compression
compress	LZ78-type, Unix *compress* utility
null	copy input to output (Unix *cat*)

Method	Relative Speed		Compression	
	Encoding	Decoding	bpc	% Remaining
pack	0.6	0.9	4.53	56.6
char	2.9	4.0	4.49	56.1
ppm	5.3	5.9	2.11	26.4
huffword	2.2	0.9	2.95	36.9
gzip-f	1.1	0.4	2.91	36.4
gzip-b	7.0	0.3	2.53	31.6
compress	1.0	0.6	3.31	41.4
null	0.2	0.2	8.00	100.0

Performance Issues: gzip

- gzip permits 9 degrees of adjustment for speed vs compression:
  - ◇ 1 = fastest, 9 = best compression
- Figures for Linux gzip compressing Moby Dick on a Pentium 366Mhz:

Setting	Time (secs)	File Size (bytes)	Compression (% remaining)
1 (fast)	0.52	581052	47.6
2	0.57	558378	45.8
3	0.76	536778	44.0
4	0.75	524590	43.0
5	1.07	507564	41.6
6	1.48	499886	41.0
7	1.60	498674	40.9
8	1.92	497990	40.8
9 (best)	2.01	497990	40.8
null	0.01	1220150	100.0