CSC310 – Information Theory Sam Roweis

Lecture 11b:

Dictionary Methods

October 24, 2005

Merits of Probabilistic Models

- N-th order Markov models and PPM models cleanly separate the *model* for symbol probabilities from the *coding* based on those probabilities.
- Such models have several advantages:
  - Coding can be nearly optimal (eg, using arithmetic coding).
  - It’s easy to try out various modeling ideas.
  - You can get very good compression, if you use a good model.
- The big disadvantage:
  - The coding and decoding involves operations for every symbol and every bit, plus possibly expensive model updates, which limits how fast these methods can be.

Merits of Dictionary Methods

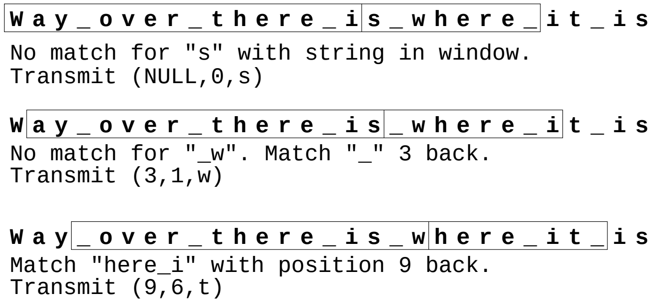
- Compression using adaptive *dictionaries* may be less elegant, but has it’s own advantages:
  - Dictionary methods can be quite fast (especially at decoding), since whole sequences of symbols are specified at once.
  - The idea that the data contain many repeated strings fits many sources quite well — eg, English text, machine-language programs, files of names and addresses.
- The main disadvantage is that compression may not be as good as a model based method:
  - Dictionaries are inappropriate for some sources — eg, noisy images.
  - Even when dictionaries work well, a good model-based method may do better — and can’t do worse, if it uses the same modeling ideas as the dictionary method.

The LZ77 Scheme

- This scheme was devised by Ziv and Lempel in 1977.
  There are many variants, including the method used by `gzip`.
- The idea of LZ77 is to use the past text as the dictionary — avoiding the need to transmit a dictionary separately.
- LZ77 uses a buffer of size W that contains the previous S characters plus the following $W - S$ characters.
- We encode up to $W - S$ characters at once by sending the following:
  - A pointer p to a past character in the buffer (or null).
    (p is an integer from 1 to S).
  - The number n of characters to take from the buffer
    (n is an integer from 0 to $W-S-1$, or maybe more).
  - The single character c that follows the string taken from the buffer.
- We chose p to maximize n (and, if possible, minimize p.)

An Example of LZ77 Coding

- Suppose we look at the past S=16 characters, and look ahead at the next 8 characters, so that W=16+8=24.
- After encoding the first 16 characters of the following string, we would proceed as follows:



```
Way_over_there_is_where_it_is
No match for "s" with string in window.
Transmit (NULL,0,s)

Way_over_there_is_where_it_is
No match for "_w". Match "_" 3 back.
Transmit (3,1,w)

Way_over_there_is_where_it_is
Match "here_i" with position 9 back.
Transmit (9,6,t)
```

- We pretend that the file is preceeded by S special symbols (e.g. spaces).

Encoding the Pointers

- If we look back S characters, we can encode the pointer back (p) in $\lceil \log_2(S+1) \rceil$ bits (the +1 is for NULL).
- If we look forward $W - S$ characters, we can encode the length of the match (n) in $\lceil \log_2(W-S) \rceil$ bits.
- If we transmit the pointer p=NULL, we don't transmit n.
- The character c after the match can be encoded in $\lceil \log_2(q) \rceil$ bits, if we have q symbols.
- If these lengths are multiples of 8, we can quickly output these codes as one or more bytes.
- An alternative: Use Huffman or arithmetic coding for the pointers. This will give better compression, but won't be as fast.

LZ77 Encoding and Decoding Speed

- Even if writing the codes for the match is fast, *finding* the longest match may be slow.
- Techniques such as hashing can speed this up, however. The `gzip` program builds a hash table for all strings of length three, then searches within the hash bucket for the next three characters to find the longest match.
- Decoding can be very fast. Reading the codes is very quick if they are take up fixed numbers of bytes. Even if we use Huffman codes, table look up on the next few bits (as in `gzip`) can be pretty fast.
- Once we have the codes, we just copy text from the buffer.

`gzip` in detail

- The algorithm used by `gzip` (and `zip`/`zlib`) is a variation of LZ77 called "deflate". It finds duplicated strings in blocks of the input data. The second occurrence of a string is replaced by a pointer to the previous occurance in the form of a pair (distance, length).
- Distances are limited to 32K bytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32K bytes, it is emitted as a sequence of literal bytes.
- Literals and match lengths are compressed with one Huffman tree; match distances are compressed with another tree. The trees are stored in a compact form at the start of each block.
- Duplicated strings are found using a hash table. All input strings of length 3 are inserted in the hash table. A hash index is computed for the next 3 bytes. If the hash chain for this index is not empty, all strings in the chain are compared with the current input string, and the longest match is selected.

The LZ78/LZW Scheme

- Ziv and Lempel introduced another, quite different scheme in 1978, in which the dictionary is kept explicitly, and contains phrases from the entire past text. The most popular variant of LZ78 is called "LZW" after its inventor, "Welch". (used in `compress` and `gif`)
- In LZW we start with a dictionary containing just the alphabet. We then encode from the current position as follows:
  – Find the longest possible match of characters following the current position with a dictionary item.
  – Transmit the index of that dictionary item.
  – Add the matched phrase *plus the character following it* to the dictionary as a new item.
  – Set the current position to the character following the match.
- Codes for dictionary indexes will have to get longer as we go, but at a fairly slow rate.

Example of LZW

Way_over_there_is_where_it_is ...

Match: W / a / y / _ / o / v / e / r / _ / t / h / er e / _ / i / s / _ / w / he / r / e / _i / t / _i / .

Trasmission: 110001 / 011101 / 110011 / 000000 ...

Add: Wa / ay / y_ / _o / ov / ve / er / r_ / _t / th / he / ere
/ e_ / _i / is / s_ / _w / wh / her / re / e_ / _it / t_ / _i

Dictionary

000000 = _
000001 = A
000010 = B
...
110100 = z

110101 = Wa
110110 = ay
110111 = y_
111000 = _o
111001 = ov

111010 = ve
111011 = er
111100 = r_
111101 = _t
111110 = th

111111 = he
1000000 = ere ← start using 6 bits!
1000001 = e_
1000010 = _i
1000011 = is
1000100 = s_
1000101 = _w
1000110 = wh

1000111 = her
1001000 = re
1001001 = e_
1001010 = _it
1001011 = t_
1001100 = _is

Theoretical properties of LZW

- In contrast to symbol/Huffman codes and arithmetic coding, the Lempel-Ziv algorithms are defined without making any mention of a probabilistic model for the source.
- Yet, given any ergodic source (i.e., one that is memoryless on sufficiently long timescales), LZW can be proven asymptotically to compress down to the entropy of the source.
- For this reason it is called a "universal" compression algorithm. (For a proof of this property, see Cover & Thomas.)

It achieves its compression, however, only by memorizing substrings that have happened so that it has a short name for them the next time they occur. The asymptotic timescale on which this universal performance is achieved may, for many sources, be unfeasibly long, because the number of typical substrings that need memorizing may be enormous. The useful performance of the algorithm in practice is a reflection of the fact that many files contain multiple repetitions of particular short sequences of characters, a form of redundancy to which the algorithm is well suited.

Block-Sorting Compression

- Block-sorting compression breaks the original input file into blocks and re-sorts the characters in each block based on the character(s) to their left.
- This re-sorting is interesting because it is reversible and because it tends to create long runs of the same character, thus yielding a new block which is more amenble to compression by straightforward adaptive algorithms. (Like the DCT or Fourier Transform.)
- `bzip` is a block-sorting compressor, which makes use of a neat trick called the Burrows-Wheeler transform for reversing the permutation.
- Such methods are not based on an explicit probabilistic model, and only work well for file larger than several thousand characters in which the context of a character is a good predictor for that character.