Lempel-Ziv

Thinking back over the lecture on Lempel Ziv, I realize that there was no real need for me to introduce Markov chains, and the next time I teach the class, I'll change the lecture. The definition of Markov chains is indeed part of the remarkable theorem that Lempel Ziv gives asymptotically optimal compression ratios for sequences, but you can explain the proof of the theorem perfectly well (better, maybe) just using sequences where each letter is chosen independently from some fixed probability distribution. So here are the notes for the lecture I should have given.

We did Huffman coding last time. Huffman coding works pretty well, in that it comes within one bit per block of the bound that Shannon gives for encoding sequences of blocks with a given set frequencies. There are some disadvantages to it. For one thing, it requires two passes through the data you wish to encode. The first pass is used for computing the frequencies of all the blocks, and the second pass for actually encoding the data. If you don't want to look at the data twice; for instance, if you're getting the data you want to encode from some kind of program, and you don't want to take the time to store it all before sending it on, this can be a problem. The Lempel Ziv algorithm constructs its dictionary on the fly, only going through the data once. This might be a problem if, for example, the first half of some document is in English and the second half is in Chinese. In this case, the dictionary constructed for the first half will be suboptimal when used on the second half. On the other hand, if you're doing Huffman coding for this document, it would also be more efficient to break it up into the two halves and use separate Huffman codes for the English half and the Chinese half.

There are many variations of Lempel Ziv around, but they all follow the same basic idea. We'll just concentrate on one of the simplest to explain and analyze, although in other versions will work somewhat better in practice. The idea is to parse the sequence into distinct phrases. The version we analyze does this greedily. Suppose, for example, we have the string

$$AABABBBABAABABBBABBABB$$

We start with the shortest phrase on the left that we haven't seen before. This will always be a single letter, in this case A:

$$A|ABABBBABAABABBBABBABB$$

We now take the next phrase we haven't seen. We've already seen A, so we take AB:

$$A|AB|ABBBABAABABBBABBABB$$

The next phrase we haven't seen is ABB, as we've already seen AB. Continuing, we get B after that:

$$A|AB|ABB|B|ABAABABBBABBABB$$

and you can check that the rest of the string parses into

$$A|AB|ABB|B|ABA|ABAB|BB|ABBA|BB$$

Because we've run out of letters, the last phrase on the end is a repeated one. That's O.K.

Now, how do we encode this? For each phrase we see, we stick it in a dictionary. The next time we want to send it, we don't send the entire phrase, but just the number of this phrase. Consider the following table

1	2	3	4	5	6	7	8	9
A	AB	ABB	B	ABA	$ABAB$	BB	$ABBA$	BB
$\emptyset A$	$1B$	$2B$	$\emptyset B$	$2A$	$5B$	$4B$	$3A$	7

The second row gives the phrases, and the third row their encodings. That is, when we're encoding the ABAB from the sixth phrase, we encode it as 5B. This maps to ABAB since the fifth phrase was ABA, and we add B to it. Here, the empty set $\emptyset$ should be considered as the 0'th phrase and encoded by 0. last piece into binary might give (without the dividers and commas, that I've inserted to make it more comprehensible)

$$,0|1,1|10,1|00,1|010,0|101,1|100,1|011,0|0111$$

We have taken the third row of the previous array, expressed all the numbers in binary (before the comma) and the letters in binary (after the comma) Note that I've mapped A to 0 and B to 1. If you had a larger alphabet, you would encode the letters by more than one bit. (In fact, you could even use a Huffman code to encode the letters if you know the frequencies of your letters.) Note also that as soon as a reference to a phrase might conceivably involve k bits (starting with the $2^k + 1$ dictionary element), I've actually used k bits, so the number of bits used before the comma keeps increasing. This ensures that the decoding algorithm knows where to put the commas and dividers. You might notice that in this case, the compression algorithm actually makes the sequence longer. This is the case for one of two reasons. Either this original sequence was too random to be compressed much, or it was too short for the asymptotic efficiency of Lempel-Ziv to start being noticeable.

How well have we encoded the string? Suppose we have broken it up into $c(n)$ phrases, where n is the length of the string. Each phrase is broken up into a reference to a previous phrase and a letter of our alphabet. The previous phrase can be represented by at most $\log_2 c(n)$ bits, since there are $c(n)$ phrases, and the letter can be represented by at most $\log_2 \alpha$ bits, where α is the size of the alphabet (in the above example, it is 2). We have thus used at most

$$c(n)(\log_2 c(n) + \log_2 \alpha)$$

bits total in our encoding.

In practice, you don't want to use to much memory for your dictionary. Thus, most implementation of Lempel-Ziv type algorithms have some maximum size for the dictionary. When it gets full, they will drop a little-used word from the dictionary and replace it by the current word. This also helps the algorithm adapt to encode messages with changing characteristics. You need to use some deterministic algorithm for which word to drop, so that both the sender and the receiver will drop the same word.

So how well does the Lempel-Ziv algorithm work? In these notes, we'll calculate two quantities. First, how well it works in the worst case, and second, how well it works in the random case where each letter of the message is chosen uniformly and independently from a probability distribution, where the ith letter appears with probability p_i. In both cases, the compression is asymptotically optimal. That is, in the worst case, the length of the encoded string of bits is $n + o(n)$. Since there is no way to compress all length-n strings to fewer than n bits, this can be counted as asymptotically optimal. In the second case, the source is compressed to length

$$H(p_1, p_2, \ldots, p_\alpha)n + o(n) = n \sum_{i=1}^{\alpha} (-p_i \log_2 p_i) + o(n),$$

which is to first order the Shannon bound. The Lempel-Ziv algorithm actually works for more general cases, including the case where the letters are produced by a random Markov chain, which I discussed in class.

Let's do the worst case analysis first. Suppose we are compressing a binary alphabet. We ask the question: what is the maximum number of distinct phrases that a string of length n can be parsed into. There are some strings which are clearly worst case strings. These are the ones in which the phrases are all possible strings of length at most k. For example, for $k = 1$, one of these strings is

$$0|1$$

with length 2. For $k = 2$, one of them is

$$0|1|00|01|10|11$$

with length 10; and for $k = 3$, one of them is

$$0|1|00|01|10|11|000|001|010|011|100|101|110|111$$

with length 34. In general, the length of such a string is

$$n_k = \sum_{j=1}^{k} j2^j$$

since it contains 2^j phrases of length j. It is easy to check that

$$n_k = (k-1)2^{k+1} + 2$$

by showing that in both expressions above for n_k, we have $n_k - n_{k-1} = k2^k$ and $n_1 = 2$. [This should be an exercise.] If we let $c(n_k)$ be the number of distinct phrases in this string of length n_k, we get that

$$c(n_k) = \sum_{i=1}^{k} 2^i = 2^{k+1} - 2$$

For n_k, we thus have

$$c(n_k) = 2^{k+1} - 2 \leq \frac{(k-1)2^{k+1}}{k-1} \leq \frac{n_k}{k-1}$$

Now, for an arbitrary length n, we can write $n = n_k + \Delta$. To get the case where $c(n)$ is largest, the first n_k bits can be parsed into $c(n_k)$ distinct phrases, containing all phrases of length at most k, and the remaining Δ bits can be parsed into into phrases of length $k+1$. This is clearly the most distinct phrases a string of length n can be parsed into, so we have that for a general string of length n, the number of phrases is at most total is

$$c(n) \leq \frac{n_k}{k-1} + \frac{\Delta}{k+1} \leq \frac{n_k + \Delta}{k-1} = \frac{n}{k-1} \leq \frac{n}{\log_2 c(n) - 3}$$

Now, we have that a general bit string is compressed to around $c(n)\log_2 c(n) + c(n)$ bits, and if we subsitute

$$c(n) \leq \frac{n}{\log_2 c(n) - 3}$$

we get

$$c(n)\log_2 c(n) + c(n) \leq n + 4c(n) = n + O(\frac{n}{\log_2 n})$$

So asymptotically, we don't use much more than n bits for compressing any string of length n. This is good: it means that the Lempel-Ziv algorithm doesn't expand any string very much. We can't hope for anything more from a general compression algorithm, as it is impossible to compress all strings of length n into fewer than n bits. So if a lossless compression algorithm compresses some strings to fewer than n bits, it will have to expand other strings to more than n bits. [Lossless here means the uncompressed string is exactly the original message.]

We now need to show that in the case of random strings, the Lempel Ziv algorithm's compression rate asymptotically approaches the entropy. Let us assume that we construct our sequence by, at each position, independently choosing letter α_i with probability p_i. This gives us a sequence

$$x = \alpha_{x(1)}\alpha_{x(2)}\alpha_{x(3)}\cdots\alpha_{x(n)}$$

We define $Q(x)$ to be the probability of seeing this sequence. That is,

$$Q(x) = \prod_{i=1}^{n} p_{x(i)}.$$

Now, suppose x is broken into distinct phrases

$$x = y_1 y_2 y_3 \ldots y_{c(n)}.$$

It is not hard to see that

$$Q(x) = \prod_{i=1}^{c(n)} Q(y_i) \tag{1}$$

Now, let's let c_l be the number of phrases y_i of length l. These are (by definition of Lempel-Ziv) all distinct. We can now prove a version of Ziv's inequality. This inequality says

$$-\log Q(x) \geq \sum_l c_l \log c_l$$

We start by rewriting Eq. 1 above

$$Q(x) = \prod_l \prod_{|y_i|=l} Q(y_i)$$

Now, let's look at the inner product. We know that, since the y_i with length l are all distinct, they are mutually independent events, so the probabilities $Q(y_i)$ sum to at most 1.

$$\sum_{|y_i|} Q(y_i) \leq 1$$

Now, there is an inequality that says that to maximize a product of k terms, given a fixed sum of these terms, the best thing to do is make all of these terms equal. There are c_l terms $Q(y_i)$ with $|y_i| = l$. Setting them all equal, we see that

$$\prod_{|y_i|=l} Q(y_i) \leq \left(\frac{1}{c_l}\right)^{c(l)}$$

Taking logs, and adding over all phrase lengths l, we get Ziv's inequality

$$-\log Q(x) \geq \sum_l c_l \log c_l$$

Suppose we have m_i occurences of letter α_i in our string x. Then,

$$Q(x) = \prod_i p_i^{m_i}$$

By the law of large numbers, there are approximately np_i occurences of letter α_i in our string x. Taking logs, we get

$$\log Q(x) \approx np_i \log p_i = nH(p_1, p_2, \ldots, p_\alpha)$$

This is the entropy, which we saw earlier was the optimal compression ratio for this probability distribution. Recall that the compression ratio of the Lempel-Ziv algorithm was approximately $c(n) \log c(n)$. Thus, all we need to do is show

$$\sum_l c_l \log c_l \approx c(n) \log c(n).$$

Let's abbreviate $c(n)$ by c. We have $\sum_l c_l = c$. This gives us the equation

$$\sum c_l \log c_l = c \log c + c \sum_l \frac{c_l}{c} \log \frac{c_l}{c}$$

The last term is $-c$ times

$$-\sum_l \frac{c_l}{c} \log \frac{c_l}{c}$$

Now, this is the entropy of the probability distribution $\pi_l = \frac{c_l}{c}$. The l are positive integers, and the expected value of l is

$$\sum_l l \frac{c_l}{c} = \frac{n}{c}$$

Now, the maximum possible of entropy for a probability distribution π_l on positive integers whose expected value is $\frac{n}{c}$ is $O(\log \frac{n}{c})$ [This should be an exercise with hints, maybe I'll do it later. But it isn't hard to see why it should be true intuitively. If the expected value is n/c, then most of the weight must be in the first $O(n/c)$ integers, and if a distribution is spread out over a sample space of size $O(n/c)$, the entropy is at most $O(\log(n/c))$.] So we get that

$$-\log Q(x) \geq c(n) \log c(n) - O(\log \frac{n}{c(n)})$$

which shows that the compression (the right side) is approximately n times the entropy (the left side), and so is optimal.

The Lempel-Ziv algorithm also works for messages that are generated by probabilistic processes with limited memory. This means that the probability of seeing a given letter may depend on the previous letters, but it can only depend on letters that are close to it. The spirit of the proof is the same, although the details get more complicated. This kind of process seems to reflect real-world sequences pretty well, in that the Lempel-Ziv family of algorithms works very well on a lot of real-world sequences.

All the compression algorithms I've talked about so far are lossless compression algorithms. This means that the reconstructed message is exactly the same as the original message. For many real-world processes, lossy compression algorithms are adequate, and these can often achieve much better compression ratios than lossless algorithms. For example, if you want to send video or music, it's generally not worth it to retain distinctions which are invisible or inaudible to our senses. Lossy compression thus gets much more complicated, because in addition to the mathematics, you have to figure out what kinds of differences can be distinguished by human eyes or ears, and then find an algorithm which ignores those kinds of differences, but doesn't lose significant differences. (And it may even be useful to have compression algorithms that reduce the quality of the signal.)