Lempel-Ziv codes

Michel Goemans

We have described Huffman coding in the previous lecture note. Huffman coding works fairly well, in that it comes within one bit per letter (or block of letters) of the bound that Shannon gives for encoding sequences of letters with a given set of 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 to compute the frequencies of all the letters, 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 to be encoded from some kind of program, and you don't have the memory to store it without encoding it first, this can be a problem. The Lempel Ziv algorithm constructs its dictionary on the fly, only going through the data once.

1 The LZ78 Algorithm

There are many variations of Lempel Ziv around. We now explain the algorithm that Lempel and Ziv gave in a 1978 paper, generally called LZ78. (An earlier algorithm, LZ77, was based on the same general idea, but is quite different in the implementation details.) The idea behind all the Lempel-Ziv algorithms is that if some text is not uniformly random; that is, if all the letters of the alphabet are not equally likely, then a substring that have already seen is more likely to appear again than a substring you haven't seen. This is certainly true for any natural language, where words will get used repeatedly, whereas strings of letters which don't appear in words will hardly ever get used.

The LZ78 algorithm works by constructing a dictionary of substrings, which we will call "phrases," that have appeared in the text. The LZ78 algorithm constructs its dictionary on the fly, only going through the data once. This means that you don't have to receive the entire document before starting to encode it. The algorithm parses the sequence into distinct phrases. We do 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 the 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
A	$1B$	$2B$	$0B$	$2A$	$5B$	$4B$	$3A$	7

The first row gives the numbers of the phrases (which you should think of as being in binary), 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 (We will later encode both 5 and B in binary to complete the encoding.) This maps to ABAB since the fifth phrase was ABA, and we add B to it. Here, the empty set $\emptyset$ is considered to be the 0'th phrase and encoded by 0. The last step of the algorithm is to encode this string into binary This gives

$$0111010010100101110010110011 1$$

To see how it works, let's insert dividers and commas (which aren't present in the real output of LZ78) 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. Note also that we've mapped 2 (referencing the second phrase) to both 10 and 010. As soon as a reference to a phrase might conceivably involve k bits (that is, starting with the $2^{k-1}+1$ dictionary element), we've used k bits to encode the phrases, 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; otherwise the receiver couldn't tell whether a '10' stood for the second phrase or was the first two bits of '100', standing for the fourth phrase.

You might notice that in this case, the compression algorithm actually made the sequence longer. This could be 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.

To decode, the decoder needs to construct the same dictionary. To do this, he first takes the binary string he receives, and inserts dividers and commas. This is straightforward. The first divider comes after one bit, the next comes after 2 bits. The next two each come after 3 bits. We then get 2^2 of length 4 bits, 2^3 of length 5 bits, 2^4 of length 6 bits, and in general 2^k of length $k+2$ bits. The phrases give the dictionary. For example, our dictionary for the above string is

$\emptyset$	0
A	1
AB	2
ABB	3
B	4
ABA	5
ABAB	6
BB	7
ABBA	8

The empty phrase $\emptyset$ is always encoded by 0 in the dictionary.

Recall that when we encoded our phrases, if we had r phrases in our dictionary (including the empty phrase $\emptyset$), we used $\lceil \log_2 r \rceil$ bits to encode the number of the phrase. (Recall $\lceil x \rceil$ is the smallest integer greater than x.) This ensures that the decoder knows exactly how many bits are in each phrase. You can see that in the example above, the first time we encoded AB (phrase 2) we encoded it as 10, and the second time we encoded it as 010. This is because the first time, we had three phrases in our dictionary, and the second time we had five.

The decoder uses the same algorithm to construct the dictionary that the encoder did; this ensures that the decoder and the encoder construct the same dictionary. The decoder knows phrases 0 through $r-1$ when he is trying to figure out what the rth phrase is, and this is exactly the information he needs to reconstruct the dictionary.

How well have we encoded the string? For an input string x, let $c(x)$ denote the number of phrases that x gets split into. Each phrase is broken up into a reference to a previous phrase and a letter of our alphabet. The previous phrase is always represented by at most $\lceil \log_2 c(x) \rceil$ bits, since there are $c(x)$ phrases, and each letter can be represented by at most $\lceil \log_2 N \rceil$ bits, where N is the size of the alphabet (in the above example, it is 2). We have thus used at most

$$c(x)(\log_2 c(x) + \log_2 N + 2)$$

bits total in our encoding.

(In practice, you don't want to use too 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 phrase from the dictionary and replace it by the current phrase. This also helps the algorithm adapt to encode messages with changing characteristics. You only need to use some deterministic algorithm for choosing 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 will analyze how well it works in the random case, where each letter of the message is chosen independently from a probability distribution with the ith letter of the alphabet having probability p_i. There's a similar worst-case calculation which shows that the algorithm will never expand the string by much. This calculation is not difficult, but is not included in these notes. It is similar to the average-case calculation. 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 binary strings to fewer than n bits (this could be an exercise), the worst-case behavior is asymptotically optimal.

For the average case calculation, we can show that the source is compressed to length

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

where $o(n)$ means a term that grows more slowly than n asymptotically. This, to leading order, is the Shannon bound, showing that we cannot do better asymptotically. The Lempel-Ziv algorithm actually works asymptotically optimally for more general cases, including cases where the letters are produced by classes of probabilistic processes where the distribution of a letter depends on the letters immediately before it.

2 Average-Case Analysis

We now need to show that in the case of random strings, the Lempel Ziv algorithm's compression rate asymptotically approaches the entropy. Let A be the alphabet and p_a be the probability of obtaining letter $a \in A$. As before, we assume that we have a "first order source"; that is, we have a random sequence of letters

$$x = X_1 X_2 \ldots X_n$$

where the X_i are independent letters from A with $\Pr(X_j = a_i) = p_i$ for all i, j. Further let us assume that $p_i \leq \frac{1}{2}$ for all i, i.e., that no letter has a large probability. These assumptions are not necessary to show that Lempel-Ziv performs well, but they do make the proof quite a bit easier.

For any particular sequence of letters

$$x = x_1 x_2 \ldots x_n,$$

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

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

In other words, if x contains n_i letters a_i for all i, then

$$P(x) = \prod_{i=1}^{N} p_i^{n_i}.$$

Note that $-\log_2 P(x)$ is closely related to the entropy of the source. Indeed in the proof of Shannon's noiseless coding theorem, we have seen that the log of the probability is close to the entropy:

$$-\log_2 P(x) \approx nH,$$

where $H = -\sum_i p_i \log_2 p_i$ is the entropy of the source.

The plan for what follows is:

1. Bound $P(x)$ in terms of $c(x)$; we want to show that messages that require many phrases (and hence are long upon encoding by Lempel-Ziv) occur with very low probability.

2. Relate $P(x)$ to the entropy.

Bounding $P(x)$ in terms of $c(x)$

Suppose the string x is broken into distinct phrases

$$x = y_1 y_2 y_3 \cdots y_{c(x)},$$

where $c(x)$ is the number of distinct phrases that x parses into. It is not hard to see that

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

Let us define C_α, $0 < \alpha \leq 1$ to be the set of phrases

$$\{y_i | \frac{\alpha}{2} < P(y_i) \leq \alpha\},$$

that is, the set of phrases with probabilities between $\alpha/2$ and α. These phrases are all different, because Lempel-Ziv parses a string into distinct phrases. Furthermore, they are all disjoint events. The only way that two phrases might not be disjoint events is if one is a subphrase of another. However, this cannot happen for two phrases in C_α, because if one phrase is a subphrase of another, the longer phrase will contain at least one letter the subphrase does not. Since the probability of every letter is at most $\frac{1}{2}$, the two phrases differ in probability by at least a factor of 2, and so cannot both be in C_α. Since any set of disjoint events sums to at most one, we have

$$\sum_{y_i \in C_\alpha} P(y_i) \leq 1.$$

Since each phrase in C_α has probability at least $\alpha/2$, and these probabilities sum to at most 1, there can be no more than $2/\alpha$ phrases in C_α.

We would like a lower bound on the number of distinct phrases $c(x)$ that a string

$$x = y_1 y_2 \cdots y_{c(x)}$$

of probability $P(x)$ can be broken into. If we fix $P(x)$, the best way to break x into phrases is to find use phrases y_i with as large probability as possible. However, we also know that the sum of the probabilities of phrases with probabilities between $\alpha/2$ and α is at most 1.

Consider the problem: put c points in the interval $(0, 1]$, so that for each subinterval $(\alpha/2, \alpha]$, the sum of the values of the points in this subinterval is at most 1, and maximize the product of the values of all the points. The solution to this problem gives an upper bound on the probability of $P(x)$ given the number of phrases $c(x)$.

One might think that the optimal placement of points is the greedy placement. Put one point at 1, two points at 1/2, four points at 1/4, and so on. This is indeed the optimal arrangement if the number of points is $2^i - 1$ for some i (difficult exercise: prove this by induction), but surprisingly, for other numbers of points, it is not optimal.[1]

However, we can still use the basic idea to bound $P(x)$. We know that there are at most two points in the interval $(\frac{1}{2}, 1]$, and that each of these is at most 1, that there are at most four points

[1]For example, placing 94 points, we find that placing points at values $1/(3*2^k)$ works better than at values $1/(2^k)$, because $1 \cdot 2^{-2} \cdot 4^{-4} \cdot 8^{-8} \cdot 16^{-16} \cdot 32^{-32} \cdot 64^{-31} \leq 1 \cdot 3^{-3} \cdot 6^{-6} \cdot 12^{-12} \cdot 24^{-24} \cdot 48^{-48}$.

in the interval $(\frac{1}{4}, \frac{1}{2}]$, and that each of these is at most $\frac{1}{2}$, and in general that there are at most 2^i points in the interval $(2^{-(i+1)}, 2^{-i}]$, and that each of these is at most 2^{-i}. Now, if 2^k is the largest power of 2 less than $c(x)-1$, then we have that

$$P(x) \leq 1^{-2}2^{-4}4^{-8}8^{-16}\ldots 2^{-(k-2)2^{k-1}}2^{(k-1)(c-2^k+2)}$$

Taking logs, and using the fact that $k = \log c + O(1)$, we have

$$\begin{aligned}
-\log_2 P(x) &\geq \sum_{i=0}^{k-2} i2^{i+1} + (k-1)(c(x) - 2^k + 2) \\
&= (k-3)2^k + 4 + (k-1)(c(x) - 2^k + 1) \\
&= (\log c(x) + O(1)\,)2^k + (\log c(x) + O(1)\,)(c(x) - 2^k + 2) \\
&= (\log c(x) + O(1)\,)(c(x) + 1) \\
&= c(x)\log c(x) + O(c(x)).
\end{aligned}$$

This is the bound we were trying to prove. We know that $nH \approx -\log_2 P(x)$, and the above argument shows that $-\log_2 P(x) \geq c(x)\log c(x) + O(c(x))$. We saw before the Lempel-Ziv compresses a sequence to $c(x)\log c(x) + O(c(x))$, so Lempel-Ziv compresses a sequence to $nH + o(n)$, asymptotically achieving the Shannon bound.

Discussion

If the highest probability letter has probability $p > \frac{1}{2}$, a more complicated version of essentially the same argument works to show that Lempel-Ziv gives optimal compression in this case as well, although a careful analysis will show that you may need larger n to obtain the asymptotic behavior here.

The Lempel-Ziv algorithm will also works for any source that is generated by a probabilistic process with limited memory. Recall that a k-order source was a source where the probability of seeing a give letter depends only on the previous k letters. The Lempel-Ziv algorithm can also show to achieve asymptotically optimal compression for a k-order source. This proof is quite a bit more complicated than the one we've seen, and requires more probability than we have used in this course. This kind of process does seem to reflect real-world sequences pretty well; the Lempel-Ziv family of algorithms works very well on many real-world sequences.