Practical Source Coding: data compression with symbol codes

Last lecture, we saw a proof of the fundamental status of the entropy as a measure of information content. We defined a data compression scheme using *fixed length block codes.* We proved that for sufficiently large N, it is possible to encode N i.i.d. variables $\mathbf{x} = (x_1, \ldots, x_N)$ into a block of $N(H(X) + \epsilon)$ bits with vanishing probability of error, whereas if we attempt to encode X^N into $N(H(X) - \epsilon)$ bits, the probability of error is virtually 1.

We thus verified the *possibility* of data compression, but the block coding defined in the proof does not give an explicit or practical algorithm. We now study practical data compression algorithms that use *variable length symbol codes.* The codes we will discuss are *lossless*, *i.e.*, they always decompress without any errors, but there is a chance that the codes may sometimes produce longer encoded strings. We will again verify the fundamental status of the entropy, proving:

> There exists a variable length encoding C of an ensemble X such that the average length of an encoded symbol is $L(C) \in [H(X), H(X) + 1)$.

We will also define a constructive procedure, the Huffman coding algorithm, which produces optimal symbol codes.

Notation for alphabets. $\mathcal{A}^N$ denotes the set of ordered N-tuples of elements from the set $\mathcal{A}$, *i.e.*, all strings of length N. The symbol $\mathcal{A}^+$ will denote the set of all strings of finite length composed of elements from the set $\mathcal{A}$.

e.g., $\{0,1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$; $\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$.

A (binary) symbol code C for an ensemble X is a mapping from the range of x, $\mathcal{A}_X = \{a_1, \ldots a_I\}$, to $\{0,1\}^+$. $c(x)$ will denote the 'codeword' corresponding to x, and $l(x)$ will denote its length, with $l_i = l(a_i)$.[1]

The *extended code* C^+ defines a mapping from $\mathcal{A}_X^+$ to $\{0,1\}^+$ as the concatentation, without punctutation:

$$c^+(x_1 x_2 \ldots x_N) = c(x_1)c(x_2)\ldots c(x_N).$$

A code $C(X)$ is uniquely decodeable if every element of $\mathcal{A}_X^+$ maps into a different string, *i.e.*,

$$\forall \mathbf{x}, \mathbf{y} \in \mathcal{A}_X^+, \mathbf{x} \neq \mathbf{y} \Rightarrow c^+(\mathbf{x}) \neq c^+(\mathbf{y}).$$

A code is called a prefix code if no codeword is a prefix of any other codeword. A prefix code is also known as an *instantaneous* code, because it can be decoded without looking ahead to subsequent codewords. The end of a codeword is immediately recognizable. A prefix code is self-punctuating, and is uniquely decodeable.

[0] These and previous lecture notes are also available by `www` or `ftp` at: `ftp://131.111.48.24/pub/mackay/info-theory/course.html`

[1] All these definitions of source codes, Huffman codes, etc., can be generalized to codes over other d-ary alphabets, but little is lost by concentrating on the binary case.

The expected length $L(C)$ of a symbol code $C(X)$ is

$$L(C) = \sum_{x \in \mathcal{A}_X} P(x) l(x).$$

Example 1: Let $\mathcal{A}_X = \{1, 2, 3, 4\}, \mathcal{P}_X = \{1/2, 1/4, 1/8, 1/8\}$, and consider the code $c(1)$=0, $c(2)$=10, $c(3)$=110, $c(4)$=111. The entropy of X is 1.75 bits, and the expected length $L(C)$ of this code is also 1.75 bits. The sequence of symbols $\mathbf{x}$=(134213) is encoded as $c^+(\mathbf{x})$ = 0110111100110. You can confirm that no other sequence of symbols $\mathbf{x}$ has the same encoding. In fact C is a *prefix code* and is therefore *uniquely decodeable.* Notice that the codeword lengths satisfy $l_i = \log_2(1/p_i)$, $p_i = 2^{-l_i}$.

Example 2: Consider the fixed length code for the same ensemble X, $c(1)$=00, $c(2)$=01, $c(3)$=10, $c(4)$=11. The expected length $L(C)$ is 2 bits.

Example 3: Consider $c(1)$=0, $c(2)$=1, $c(3)$=00, $c(4)$=11. The expected length $L(C)$ is 1.25 bits, which is less than $H(X)$. But the code is not uniquely decodeable. The sequence $\mathbf{x}$=(134213) encodes as 000111000, which can also be decoded as (312431), for example.

Example 4: Consider the code $c(1)$=0, $c(2)$=01, $c(3)$=011, $c(4)$=111. The expected length $L(C)$ of this code is 1.75 bits. The sequence of symbols $\mathbf{x}$=(134213) is encoded as $c^+(\mathbf{x})$ = 0011111010011. C is uniquely decodeable but is not a prefix code. When we receive '00', it is possible that $\mathbf{x}$ could start '11', '12' or '13'. Once we have received '001111', the second symbol is still ambiguous, as $\mathbf{x}$ could be '124...' or '134...'. Eventually a unique decoding crystallizes.

Fortunately, it turns out that to make an optimal symbol code we can restrict attention to prefix codes.

What limit is imposed by unique decodeability?

We now ask the question, given a list of positive integers $\{l_i\}$, does there exist a uniquely decodeable code with those integers as its codeword lengths?

Kraft-McMillan inequality. For any uniquely decodeable code C over the binary alphabet $\{0,1\}$, the codeword lengths must satisfy:

$$\sum_i 2^{-l_i} \leq 1.$$

Conversely, given a set of codeword lengths that satisfy this inequality, there exists a uniquely decodeable prefix code with these codeword lengths.

Proof: Define $S = \sum_i 2^{-l_i}$. Consider the quantity

$$S^N = \left[\sum_i 2^{-l_i}\right]^N = \sum_{i_1=1}^{I} \sum_{i_2=1}^{I} \ldots \sum_{i_N=1}^{I} 2^{-\left(l_{i_1} + l_{i_2} + \ldots l_{i_N}\right)}$$

The quantity in the exponent, $(l_{i_1} + l_{i_2} + \ldots l_{i_N})$, is the length of the encoding of the string $\mathbf{x} = a_{i_1} a_{i_2} \ldots a_{i_N}$. For every string $\mathbf{x}$ of length N, there is one term in the above sum. Introduce an array A_L that counts how many strings

$\mathbf{x}$ have encoded length L. Then, defining $l_{\min} = \min_i l_i$ and $l_{\max} = \max_i l_i$:

$$S^N = \sum_{L=Nl_{\min}}^{Nl_{\max}} 2^{-L} A_L.$$

Now assume C is uniquely decodeable, so that for all $\mathbf{x} \neq \mathbf{y}$, $c^+(\mathbf{x}) \neq c^+(\mathbf{y})$. Concentrate on the $\mathbf{x}$ that have encoded length L. There are a total of 2^L distinct bit strings of length L, so it must be the case that $A_L \leq 2^L$. So

$$S^N = \sum_{L=Nl_{\min}}^{Nl_{\max}} 2^{-L} A_L \leq Nl_{\max}.$$

This result is true for any N. Thus $S^N \leq l_{\max} N$ for all N. Therefore $S \leq 1$. Q.E.D.
The proof of the converse is left as an optional exercise.

Lower bound on expected length. The expected length $L(C)$ of a uniquely decodeable code is bounded below by $H(X)$.

Proof: We define the implicit probabilities $q_i \equiv 2^{-l_i}/c$, where $z = \sum_{i'} 2^{-l_{i'}}$, so that $l_i = \log 1/q_i - \log z$. We then use Gibbs' inequality, $\sum_i p_i \log 1/q_i \geq \sum_i p_i \log 1/p_i$, with equality if $q_i = p_i$, and the Kraft-McMillan inequality $z \leq 1$:

$$\begin{aligned} L(C) &= \sum_i p_i l_i = \sum_i p_i \log 1/q_i - \log z \\ &\geq \sum_i p_i \log 1/p_i - \log z \\ &\geq H(X). \end{aligned}$$

The equality $L(C) = H(X)$ is achieved only if the Kraft-McMillan equality $z = 1$ is satisfied, and if the code lengths satisfy $l_i = \log(1/p_i)$.

This is an important result so let's say it again:

Optimal source code lengths. The expected length is minimized only if the code lengths are:

$$l_i^* = \log(1/p_i).$$

Theorem 1 *For an ensemble X there exists a prefix code with*

$$H(X) \leq L(C) < H(X) + 1.$$

Proof (sketch): We set the code lengths as close as possible to the optimum lengths:

$$l_i = \lceil \log(1/p_i) \rceil$$

where $\lceil l^* \rceil$ denotes the smallest integer greater than or equal to l^*. We check that there is a prefix code with these lengths by confirming that the Kraft-McMillan inequality is satisfied.

Then we confirm $L(C) = \sum_i p_i \lceil \log(1/p_i) \rceil < \sum_i p_i (\log(1/p_i) + 1) < H(X) + 1$.

Note in passing what the average message length will be when we use the 'wrong' code. If the 'true probability' is $\{p_i\}$ and we use a code with lengths l_i that satisfy the Kraft-McMillan inequality, we can view those lengths as defining an implicit probabilistic model $q_i = 2^{-l_i}$. The average length is $L(C) = H(X) + \sum_i p_i \log p_i/q_i$, *i.e.*, it exceeds the entropy by the Kullback-Leibler distance $D_{\mathrm{KL}}(p||q)$.

How to do it: the Huffman coding algorithm

We construct the code backwards starting from the tails of the codewords.[2]

> Take the two least probable symbols in the alphabet. These two symbols will be given the longest codewords, which will have equal length, and differ only in the last digit. Combine these two symbols into a single symbol, and repeat.

Example: Let $\mathcal{A}_X = \{1, 2, 3, 4, 5\}$
with $\mathcal{P}_X = \{0.25, 0.25, 0.2, 0.15, 0.15\}$.

x					
1	0.25	0.25	0.25 0	0.55 0	1.0
2	0.25	0.25 0	0.45 1	0.45 1	
3	0.2	0.2 1			
4	0.15 0	0.3	0.3 1		
5	0.15 1				

The Huffman codes are then obtained by concatenating the symbols in reverse order: $C = \{00, 10, 11, 010, 011\}$.

The proof that there is no better symbol code for a source than the Huffman code is left as an optional exercise.

Disadvantages of the Huffman code

The Huffman algorithm produces an optimal symbol code for an ensemble, but this is not the end of the story.

Changing ensemble. If we wish to communicate a sequence of outcomes from one unchanging ensemble, then a Huffman code may be convenient. But often the appropriate ensemble changes adaptively. If, for example, we are compressing text, then the symbol frequencies will be different in one context from another. And furthermore, our knowledge of these context-dependent symbol frequencies will also adapt as we accumulate statistics on the text source.

Huffman codes do not relate with any elegance to adapting ensemble probabilities. One brute-force approach would be to recompute the Huffman code every time a new probability over symbols is introduced. Another attitude is to deny the option of adaptation, and instead to run through the entire file in advance and compute good probability distributions which will then remain fixed throughout transmission. The code itself has to first be communicated in this scenario. Such a technique is not only cumbersome and restrictive, it is also sub-optimal, since the initial message specifying the code and the document itself are partially redundant—knowing the algorithm for defining the

[2] How not to do it: one might try to roughly split the set $\mathcal{A}_X$ in two, and continue bisecting the subsets so as to define a binary tree from the top. This is how the *Shannon-Fano code* is constructed, but it is not necessarily optimal; it achieves $L(C) \leq H(X) + 2$.

code given a document, one can deduce what the initial message has to be. This technique is therefore wasteful of bits.

The extra bit. An equally serious problem with Huffman codes is the innocuous 'extra bit' relative to the ideal average length of $H(X)$. A Huffman code incurs an overhead of between 0 and 1 bits per symbol. If $H(X)$ were large, then this would be an unimportant fractional increase. But for many applications, the entropy may be as low as one bit per symbol, or even less, so that the '+1' may dominate the encoded file length. Consider English text: in some contexts, long strings of characters may be highly predictable, given a simple model of the language. For example, in the context '`strings of ch`', one might predict the next nine symbols '`aracters `' with a probability of 0.99 each. A traditional Huffman code would be obliged to use one bit per character, making a total of nine bits where virtually no information is being conveyed.

A traditional patch-up of Huffman codes uses them to compress *blocks* of symbols from 'extended sources' X^N. The overhead per symbol goes down as $1/N$, so for sufficiently large blocks, the problem of the extra bit may be removed, but only at the expense of losing the elegant instantaneous decodeability of the simple Huffman coding. A further problem is that it may not be appropriate to model successive symbols as coming i.i.d. from a single ensemble X. As we already argued, any decent model for text will assign a probability over symbols that changes from letter to letter.

Huffman codes, therefore, although widely trumpeted as 'optimal', have many defects for practical purposes.

These defects are rectified by *Arithmetic coding*, which dispenses with the restriction that each symbol must translate into an integer number of bits.

Appendix—proof of Gibbs' inequality

Convex function. A function f(r) is *convex* over (a, b) if for all $r_1, r_2 \in (a, b)$ and $0 \leq \lambda \leq 1$,

$$f(\lambda r_1 + (1-\lambda) r_2) \leq \lambda f(r_1) + (1-\lambda) f(r_2).$$

A function f is strictly convex if the equality holds only for λ=0 and λ=1. Some strictly convex functions are r^2, e^r, $\log(1/r)$ for $r > 0$, and $r \log r$ for $r > 0$.

Jensen's inequality. If f is a convex function and r is a random variable then:

$$E\left[f(r)\right] \geq f\left(E[r]\right),$$

where E denotes expectation. If f is strictly convex and $E\left[f(r)\right] = f\left(E[r]\right)$, then r is a constant (with probability 1).

The relative entropy *or* **Kullback-Leibler distance** between two probability distributions $p(x)$ and $q(x)$ that are defined over the same $x \in \mathcal{A}_X$ is

$$D_{\mathrm{KL}}(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)}.$$

Gibbs' inequality. The relative entropy satisfies $D_{\mathrm{KL}}(p||q) \geq 0$ with equality only if p=q.

Proof: Use Jensen's inequality, with $r \equiv q/p$ and $f(r) \equiv \log \frac{1}{r}$: $E\left[\log 1/(q/p)\right] \geq \log\left(1/E[q/p]\right) = \log\left(1/[\sum_i p_i q_i/p_i]\right) = 0$, with equality if q_i/p_i = constant.

Corollary: $\sum_x p(x) \log(1/q(x)) \geq \sum_x p(x) \log(1/p(x))$.