CSC 310: Information Theory

University of Toronto, Fall 2011

Instructor: Radford M. Neal

Week 6

Where Do the Probabilities Come From?

So far, we've assumed that we "just know" the probabilities of the symbols, $p_1, \ldots, p_I$. Note: The transmitter and the receiver must both know the *same* probabilities.

This isn't realistic. For instance, if we're compressing black-and-white images, there's no reason to think we know beforehand the fraction of pixels in the transmitted image that are black.

But could we make a good guess? That might be better than just assuming equal probabilities. Most fax images are largely white, for instance. Guessing $P(\text{White}) = 0.9$ may usually be better than using $P(\text{White}) = 0.5$.

The Penalty for Guessing Wrong.

Suppose we use a code that would be optimal if the symbol probabilities were $q_1, \ldots, q_I$, but the real probabilities are $p_1, \ldots, p_I$. How much does this cost us?

Assume we use large blocks (ie, N'th extension with large N), or use arithmetic coding — so that an optimal code compresses to nearly the entropy, given the assumed probabilities.

We can compute the difference in expected code length between an optimal code based on $q_1, \ldots, q_I$ and an optimal code based on the real probabilities, $p_1, \ldots, p_I$, as follows:

$$\sum_{i=1}^{I} p_i \log(1/q_i) \;-\; \sum_{i=1}^{I} p_i \log(1/p_i) \;=\; \sum_{i=1}^{I} p_i \log(p_i/q_i)$$

This is called the *relative entropy* of $\{p_i\}$ and $\{q_i\}$. As we proved earlier (in week 3 lectures), it can never be negative.

Why Not Estimate the Probabilities, Then Send Them With the Data?

One way to handle unknown probabilities is to have the transmitter *estimate* them, and then send these probabilities along with the compressed data, so that the receiver can uncompress the data correctly.

Example: We might estimate the probabilitiy that a pixel in a black-and-white image is black by the *fraction* of pixels in the image we are sending that are black.

One problem: We need some code for sending the estimated probabilities. How do we decide on that? We will need to guess the probabilities for the different probabilities.

Another problem: How precise should our probabilities be? Sending very precise probabilities may take a lot of bits, but result in compression that's not much better than with less precise probabilities.

Sending and Using Estimated Probabilities Isn't Optimal

Such a scheme may sometimes be a pragmatic solution, but it can't possibly be optimal, because the resulting code isn't *complete* — ie, not all sequences of code bits are possible. As discussed earlier, a prefix code is not complete if some nodes in its tree have only one child.

Suppose we send a 3-by-5 black-and-white image by first sending the number of black pixels (0 to 15) and then the 15 pixels themselves, as one block, using probabilities estimated from the count sent.

Some messages will not be possible. For example:

4 ○ ● ● ○ ○
● ● ● ● ○
○ ○ ● ● ●

This can't happen, since the count of 4 is inconsistent with the image that follows, which has 9 black pixels.

Adaptive Models

We can do better using an *adaptive* model, which continually re-estimates probabilities using counts of symbols in the *earlier* part of the message.

We need to avoid giving any symbol zero probability, since its "optimal" codeword length would then be $\log(1/0) = \infty$. One "kludge": Just add one to all the counts.

Example: We might encode the 107th pixel in a black-and-white image using the count of how many of the previous 106 pixels are black.

If 13 of these 106 pixels were black, we encode the 107th pixel using

$$P(\text{Black}) = (13+1)/(106+2) = 0.1308$$

Changing probabilities like this is easy with arithmetic coding, harder with Huffman codes, especially if we encode blocks of symbols.

Why This Isn't Just a Kludge

Adding one to all the counts may seem like a horrible kludge for avoiding probabilities of zero. But it is actually one of the methods that can be justified by the statistical theory of *Bayesian inference*.

Bayesian inference uses probability to represent uncertainty about anything — not just which symbol will be sent next, but also what the *probabilities* of the various symbols are.

For our black-and-white image example, only one probability is unknown: $p_1 = P(\text{Black})$. (Since $P(\text{White}) = 1 - p_1$.)

The system designer starts by selecting a *prior distribution* for p_1, that expresses what is known about p_1 *before* seeing the image that is to be encoded.

The Posterior Distribution

Suppose the encoder has seen and already encoded n_B black pixels and n_W white pixels. What should the encoder (and the decoder, following the same algorithm) use as the probability that the next pixel is black?

To find this, we first find the *posterior distribution* for the unknown probability, p_1, which we get using Bayes' Rule:

$$
\begin{aligned}
&P(p_1 \,|\, \text{pixels already encoded}) \\
&\quad = \frac{P(\text{pixels already encoded} \,|\, p_1) P(p_1)}{\int_0^1 P(\text{pixels already encoded} \,|\, p_1) P(p_1) dp_1}
\end{aligned}
$$

$P(p_1)$ is the prior probability density for p_1 (fixed by the system designer).

$P(\text{pixels observed} \,|\, p_1)$ is called the *likelihood*. It captures what is learned about p_1 from the data. For this example:

$$
P(\text{pixels already encoded} \,|\, p_1) \;=\; p_1^{n_B} \, (1 - p_1)^{n_W}
$$

The Predictive Distribution

To make a prediction, the encoder finds the *average* probability of a symbol with respect to its posterior distribution.

If the encoder has seen and encoded n_B black pixels and n_W white pixels, and the system designer chose the *uniform* prior for p_1, for which $P(p_1) = 1$, the prediction will be that next pixel has the following probability of being black:

$$P(\text{Black}) \;=\; \int_0^1 p_1 \, \frac{p_1^{n_B}\,(1-p_1)^{n_W}}{\int_0^1 p_1^{n_B}\,(1-p_1)^{n_W}\,dp_1} dp_1$$

A useful fact:

$$\int_0^1 p^a (1-p)^b dp \;=\; a!b!\,/\,(a+b+1)!$$

Using this, we find that

$$\begin{aligned} P(\text{Black}) \;&=\; \frac{(n_B+n_W+1)!}{n_B!\,n_W!}\,\frac{(n_B+1)!\,n_W!}{(n_B+n_W+2)!} \\ &=\; (n_B+1)/(n_B+n_W+2) \end{aligned}$$

Models Assign Probabilities to Sequences of Symbols

Any way of producing predictive probabilities for each symbol in turn will also assign a probability to every *sequence* of symbols. (We'll suppose sequences are terminated by a special EOF symbol.)

We just multiply together the predictive probabilities as we go.

For example, the string "CAT" has probability

$$
\begin{aligned}
& P(X_1 = \text{`C'}) \\
& \quad \times\ P(X_2 = \text{`A'} \,|\, X_1 = \text{`C'}) \\
& \quad \times\ P(X_3 = \text{`T'} \,|\, X_1 = \text{`C'},\ X_2 = \text{`A'}) \\
& \quad \times\ P(X_4 = \text{EOF} \,|\, X_1 = \text{`C'},\ X_2 = \text{`A'},\ X_3 = \text{`T'})
\end{aligned}
$$

The probabilities above are the ones used to code each individual symbol.

With an optimal coding method (applied to the whole message or at least to large blocks), the number of bits used to encode the entire sequence will be close to $\log_2$ of one over its probability.

Probabilities of Sequences with the Laplace Model

The general form of the "add one to all the counts" method uses the following predictive distributions:

$$P(X_n = a_i) \;=\; \frac{1 + \text{Number of earlier occurrences of } a_i}{I + n - 1}$$

where I is the size of the source alphabet. This is called "Laplace's Rule of Succession".

So the probability of a sequence of n symbols is

$$\frac{(I-1)!}{(I+n-1)!} \prod_{i=1}^{I} n_i!$$

where n_i is the number of times a_i occurs in the sequence.

It's much easier to code one symbol at a time (using arithmetic coding) than to encode a whole sequence at once, but we can see from this what the model is really saying about which sequences are more likely.

Models With Multiple Contexts

So far, we've looked at models in which we assume that the symbols would be *independent*, if we knew what their probabilities were.

If we don't know the probabilities, our predictions do depend on previous symbols, but the symbols are still "exchangeable" — their order doesn't matter.

Very often, this isn't right: The probability of a symbol may depend on the *context* in which it occurs — eg, what symbol precedes it.

Example: "U" is much more likely after "Q" (in English), than after another "U".

Probabilities may also depend on position in the file, though modeling this is less common.

Example: Executable program files may have machine instructions at the beginning, and symbols to help with debugging at the end.

Markov Sources

An *K-th order Markov source* is one in which the probability of a symbol depends on the preceding K symbols.

The probability of a sequence of symbols, $X_1, X_2, \ldots, X_n$ from such a source with $K = 2$ is as follows (assuming we know all the probabilities):

$$\begin{aligned}
P(X_1 = a_{i_1}, X_2 = a_{i_2}, \ldots, X_n = a_{i_n}) \\
= \quad & P(X_1 = a_{i_1}) \times P(X_2 = a_{i_2} \mid X_1 = a_{i_1}) \\
& \times P(X_3 = a_{i_3} \mid X_1 = a_{i_1}, X_2 = a_{i_2}) \\
& \times P(X_4 = a_{i_4} \mid X_2 = a_{i_2}, X_3 = a_{i_3}) \\
& \qquad \ldots \\
& \times P(X_n = a_{i_n} \mid X_{n-2} = a_{i_{n-2}}, X_{n-1} = a_{i_{n-1}}) \\
= \quad & P(X_1 = a_{i_1}) \times P(X_2 = a_{i_2} \mid X_1 = a_{i_1}) \\
& \times M(i_1, i_2, i_3) M(i_2, i_3, i_4) \cdots M(i_{n-2}, i_{n-1}, i_n)
\end{aligned}$$

Here, $M(i, j, k)$ is the probability of symbol a_k when the preceding two symbols were a_i and a_j.

Adaptive Markov Models

Some sources may really be Markov of some order K, but usually not.

We can nevertheless use a Markov *model* for a source as the basis for data compression — it won't be perfect, but it may be pretty good.

Usually, we don't know the "transition probabilities", so we estimate them adaptively, using past frequencies. Eg, for $K = 2$, we accumulate frequencies in each context, $F(i, j, k)$, and then use probabilities

$$M(i, j, k) = F(i, j, k) / \sum_{k'} F(i, j, k')$$

After encoding symbol a_k in context a_i, a_j, we increment $F(i, j, k)$.

A K-th order Markov model has to handle the first $K-1$ symbols specially. One approach: Imagine that there are K symbols before the beginning with some special value (eg, space).

Markov Models of Order 0, 1, and 2 on English Text

Compression with adaptive arithmetic coding, Latex of different sizes:

Order 0 model:

Uncompressed file size	Compressed file size	Compression factor	Bits per character
2344	1431	1.64	4.88
20192	12055	1.67	4.78
235215	137284	1.71	4.67

Order 1 model:

Uncompressed file size	Compressed file size	Compression factor	Bits per character
2344	1750	1.34	5.97
20192	11490	1.76	4.55
235215	114494	2.05	3.89

Order 2 model:

Uncompressed file size	Compressed file size	Compression factor	Bits per character
2344	2061	1.14	7.03
20192	13379	1.51	5.30
235215	111408	2.11	3.79

How Large an Order Should be Used?

We can see a problem with these results.

A Markov model of high order works well with long files, in which most of the characters are encoded after good statistics have been gathered.

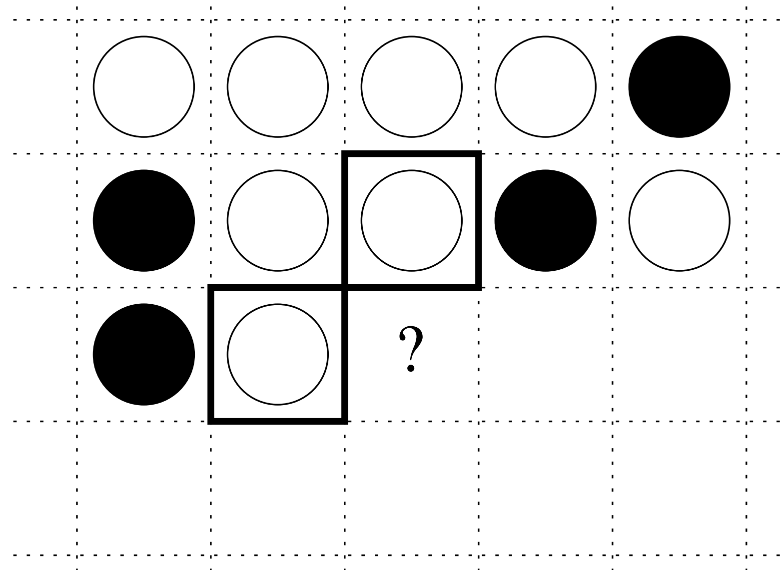
But for small files, high-order models don't work well — most characters occur in contexts that have been seen only a few times before, or never before.

For the smallest file, the zero-order model with only one context was best, even though we know that English has strong dependencies between characters!

Example: Adaptively Compressing Black-White Images

Suppose we have images (of some fixed dimensions) in which pixels are either black or white. We may expect large regions of white pixels and large regions of black pixels, and wish to use that knowledge to compress them.

But how large will the regions be? We probably don't know, so we use an adaptive scheme, with contexts defined by pixels above and to the left:



The Encode Program

```
/* Initialize model. */

for (a = 0; a<2; a++) {
    for (l = 0; l<2; l++) {
        freq0[a][l] = 1;                /* Set frequencies of 0's */
        freq1[a][l] = 1;                /* and 1's to be equal.   */
    }
}

/* Encode image. */

for (i = 0; i<Height; i++) {
    for (j = 0; j<Width; j++) {
        a = i==0 ? 0 : image[i-1][j];  /* Find current context.  */
        l = j==0 ? 0 : image[i][j-1];
        encode_bit(image[i][j],         /* Encode pixel.          */
                   freq0[a][l],freq1[a][l]);
        if (image[i][j]) {              /* Update frequencies for */
            freq1[a][l] += 1;           /* this context.          */
        }
        else {
            freq0[a][l] += 1;
        }
        if (freq0[a][l]+freq1[a][l]>Freq_full) { /* Avoid huge   */
            freq0[a][l] = (freq0[a][l]+1) / 2;  /* frequencies  */
            freq1[a][l] = (freq1[a][l]+1) / 2;
        }
    }
}
```

The Decode Program

```
/* Initialize model. */

for (a = 0; a<2; a++) {
    for (l = 0; l<2; l++) {
        freq0[a][l] = 1;                /* Set frequencies of 0's */
        freq1[a][l] = 1;                /* and 1's to be equal.   */
    }
}

/* Decode and write image. */

for (i = 0; i<Height; i++) {
    for (j = 0; j<Width; j++) {
        a = i==0 ? 0 : image[i-1][j];   /* Find current context.  */
        l = j==0 ? 0 : image[i][j-1];
        image[i][j] =                   /* Decode pixel.          */
          decode_bit(freq0[a][l],freq1[a][l]);
        printf("%c%c",image[i][j] ? '#' : '.',
                      j==Width-1 ? '\n' : ' ');
        if (image[i][j]) {              /* Update frequencies for */
            freq1[a][l] += 1;           /* this context.          */
        }
        else {
            freq0[a][l] += 1;
        }
        if (freq0[a][l]+freq1[a][l]>Freq_full) { /* Avoid huge   */
            freq0[a][l] = (freq0[a][l]+1) / 2;  /* frequencies  */
            freq1[a][l] = (freq1[a][l]+1) / 2;
        }
    }
}
```