Lecture 5:

Sam Roweis

September 26, 2005

Entropy as a measure of surprise

- What does information do? It removes uncertainty. Information Conveyed = Uncertainty Removed = Surprise Yielded.
- How should we quantify information/uncertainty/surprise? Here are some properties *any* function $h(p(event))$ should possess:
  1. $h(1) = 0$ (no surprise for certain events)
  2. $h(0) = \infty$ (infinite surprise for impossible events)
  3. $p_i > p_j \Rightarrow h(p_i) < h(p_j)$ (higher prob. means less surprise)
  4. x, y independent $\Rightarrow h(p(x_i \& y_j)) = h(p(x_i)) + h(p(x_j))$. (surprise is additive for independent events)
  5. $h(\cdot)$ is continuous (small change in prob, small change in surprise)
- What functions $h(\cdot)$ satisfy all the requirements above? The *only* consistent solution is $h(p) = -a \log_b(p)$. By convention, we chose $a = 1, b = 2$ which sets the units to bits.

Reminder: Entropy and Optimal Codes

- Last class we introduced the *entropy* $H = \sum_i p_i \log(1/p_i)$ of a source which provides a hard lower bound on the average per-symbol encoding length for *any* decodable code.
- We also learned about Huffman's Algorithm which constructs an "optimal" *symbol code* for any source.
- But Huffman Codes may be as much as one bit worse than the entropy on average. To get closer to lower limit of the entropy, we are going to consider *block coding* in which we encode several adjacent source symbols together (called the *extension* of a source).
- This introduces some decoding delay, since we can't decode any any member of the block until the code for the entire block is received, but it gets us closer to the entropy in performance.
- First, let's talk a bit more about entropy, then let's see why Huffman codes are optimal, then we can go back to block coding.

Proving that Binary Huffman Codes are Optimal

- We can prove that the binary Huffman code procedure produces optimal codes by induction on the number of source symbols, I.
- For $I = 2$, the code produced is obviously optimal — you can't do better than using one bit to code each symbol.
- For $I > 2$, we assume that the procedure produces optimal codes for *any* alphabet of size $I - 1$ (with *any* symbol probabilities), and then prove that it does so for alphabets of size I as well.
- So, to start with, suppose the Huffman procedure produces optimal codes for alphabets of size $I - 1$.
- Let L be the expected codeword length of the code produced by the procedure when it is used to encode the symbols $a_1, \ldots, a_I$, having probabilities $p_1, \ldots, p_I$. Without loss of generality, let's assume that $p_i \geq p_{I-1} \geq p_I$ for all $i \in \{1, \ldots, I-2\}$.

The Induction Step

- The recursive call in the procedure will have produced a code for symbols $a_1, \ldots, a_{I-2}, a'$, having probabilities $p_1, \ldots, p_{I-2}, p'$, with $p' = p_{I-1} + p_I$. By the induction hypothesis, this code is optimal. Let its average length be L'.
- Suppose some other instantaneous code for $a_1, \ldots, a_I$ had expected length less than L. We can modify this code so that the codewords for a_{I-1} and a_I are "siblings" (ie, they have the forms $z0$ and $z1$) while keeping its average length the same, or less.
- Let the average length of this modified code be $\widehat{L}$, which must also be less than L. From this modified code, we can produce another code for $a_1, \ldots, a_{I-2}, a'$. We keep the codewords for $a_1, \ldots, a_{I-2}$ the same, and encode a' as z.
  Let the average length of this code be $\widehat{L}'$.

Won't top-down splitting work just as well?

- Shannon & Fano had another technique for constructing a prefix code, other than rounding up $\log 1/p_i$ to get l_i.
- They arranged the source symbols in order from most probable to least probable, and then divided into two sets whose total probabilities are *as close as possible to being equal*.
- All symbols then have the first digits of their codes assigned; symbols in the first set receive "0" and symbols in the second set receive "1". As long as any sets with more than one member remain, the same process is repeated on those sets, to determine successive digits of their codes.
- When a set has been reduced to one symbol, of course, this means the symbol's code is complete and furthermore it is guaranteed to not form the prefix of any other symbol's code.

The Induction Step (Conclusion)

- We now have two codes for $a_1, \ldots, a_I$ and two for $a_1, \ldots, a_{I-2}, a'$. The average lengths of these codes satisfy the following equations:

$$L = L' + p_{I-1} + p_I$$
$$\widehat{L} = \widehat{L}' + p_{I-1} + p_I$$

  Why? The codes for $a_1, \ldots, a_I$ are like the codes for $a_1, \ldots, a_{I-2}, a'$, except that one symbol is replaced by two, whose codewords are one bit longer. This one additional bit is added with probability $p' = p_{I-1} + p_I$.
- Since L' is the optimal average length, $L' \leq \widehat{L}'$. From these equations, we then see that $L \leq \widehat{L}$, which contradicts the supposition that $\widehat{L} < L$.
- The Huffman procedure therefore produces optimal codes for alphabets of size I. By induction, this is true for all I.

Shannon-Fano top-down splitting

- Example:
  Symbol probabilities
  $(p_1 = .35, p_2 = .17, p_3 = .17, p_4 = .16, p_5 = .15)$.
  First split $(.35, .17) \to 0; (.17, .16, .15) \to 1.$
  Second split: $.35 \to 00; .17 \to 01.$
  Third split: $.17 \to 10; (.16, .15) \to 11.$
  Final split: $.16 \to 110; .15 \to 111.$

Top-down splitting is sub-optimal

- The above algorithm works, and it produces fairly efficient variable-length encodings.
- When the two smaller sets produced by a partitioning happen to be exactly equal in probabilitity, then the one bit of information used to distinguish them is used most efficiently.
- Unfortunately, Shannon-Fano top-down splitting does not always produce optimal prefix codes: the code we obtained above was $\{00, 01, 10, 110, 111\}$ which has average $L = 2.31$ bits/symbol.
- Huffman's algorithm produces a code of $\{0, 111, 110, 101, 100\}$ which has average $L = 2.30$ bits/symbol.
- Why, intuitively, does Huffman's algorithm work?

Another Way to Compress Down to the Entropy

- We get a similar result by supposing that we will always encode N symbols into a block of exactly NR bits.
  Can we do this in a way that is very likely to be decodable?
- Yes, for large values of N. The Law of Large Numbers (LLN) tells us that the sequence of symbols to encode, $a_{i_1}, \ldots, a_{i_N}$, is very likely to be a "typical" one, for which

$$\frac{1}{N} \log_2(1/(p_{i_1} \cdots p_{i_N})) \;=\; \frac{1}{N} \sum_{j=1}^{N} \log_2(1/p_{i_j})$$

  is very close to the expectation of $\log_2(1/p_i)$, which is the entropy, $H(X) = \sum_i p_i \log_2(1/p_i)$. (See Section 4.3 of MacKay's book.)
- So if we encode all the sequences in this *typical set* in a way that can be decoded, the code will almost always be uniquely decodable.

Shannon's Noiseless Coding Theorem

- By using extensions of the source, we can compress *arbitrarily close to the entropy*! Formally:

  > For any desired average length per symbol, R, that is greater than the binary entropy, $H(X)$, there is a value of N for which a uniquely decodable binary code for X^N exists that has expected length less than NR.

- Consider coding the N-th extension of a source whose symbols have probabilities $p_1, \ldots, p_I$, using an binary Shannon-Fano code.
- The Shannon-Fano code for blocks of N symbols will have expected codeword length, L_N, no greater than $1 + H(X^N) = 1 + NH(X)$.
- The expected codeword length per original source source symbol will therefore be no greater than $\frac{L_N}{N} \;=\; \frac{1+NH(X)}{N} \;=\; H(X) + \frac{1}{N}$.
- By choosing N to be large enough, we can make this as close to the entropy, $H(X)$, as we wish.

How Big is the Typical Set?

- Let's define "typical" sequences as ones where

$$(1/N) \log_2(1/(p_{i_1} \cdots p_{i_N})) \;\le\; H(X) + \eta/\sqrt{N}$$

  The probability of any such typical sequence will satisfy

$$p_{i_1} \cdots p_{i_N} \;\ge\; 2^{-NH(X) - \eta\sqrt{N}}$$

- We scale the margin allowed above $H(X)$ as $1/\sqrt{N}$ since that's how the standard deviation of an average scales. LLN (Chebychev's inequality) then tells us that most sequences will satisfy this condition, for some large enough value of η.
- The total probability for all such sequences can't be greater than one, so the number of "typical" sequences can't be greater than

$$2^{NH(X)+\eta\sqrt{N}}$$

- We will be able to encode these sequences in NR bits if $NR \ge NH(X) + \eta\sqrt{N}$. If $R > H(X)$, this will be true if N is sufficiently large.

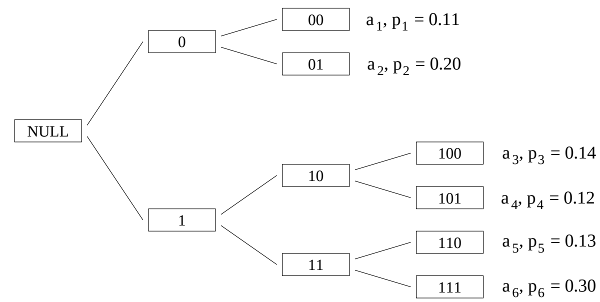
Some Observations about Instantaneous Codes

- Suppose we have an instantaneous code for symbols $a_1, \ldots, a_I$, with probabilities $p_1, \ldots, p_I$ and codeword lengths $l_1, \ldots, l_I$.
- Under each of the following conditions, we can find a better instantaneous code, i.e. one with smaller expected codeword length:

1. If $p_1 < p_2$ and $l_1 < l_2$: Swap the codewords for a_1 and a_2.
2. If there is a codeword of the form xby, where x and y are strings of zero or more bits, and b is a single bit, but there are no codewords of the form $xb'z$, where z is a string of zero or more bits, and $b' \neq b$:
   Change all the codewords of the form xby to xy. (This improves things if none of the p_i are zero, and never makes things worse.)

Continuing to Improve the Example

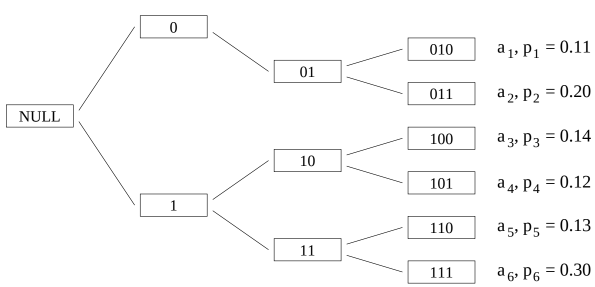
- The result is the code shown below:



- Now we note that a_6, with probability 0.30, has a longer codeword than a_1, which has probability 0.11. We can improve the code by swapping the codewords for these symbols.

The Improvements in Terms of Trees

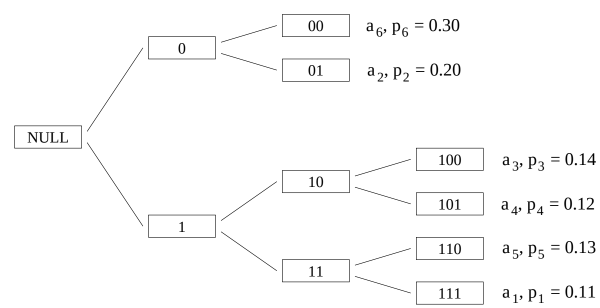
- We can view these improvements in terms of the trees for the codes. Here's an example:



- Two codewords have the form $01\ldots$ but none have the form $00\ldots$ (ie, there's only one branch out of the 0 node).
- We can therefore improve the code by deleting the surplus node.

The State After These Improvements

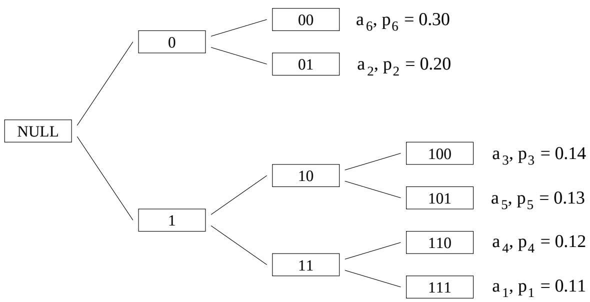
- Here's the code after this improvement:



- In general, after such improvements:
  The most improbable symbol will have the longest codeword and there will be at least one other codeword of this length — its "sibling" in the tree. The second-most improbable symbol will also have a codeword of the longest length.

A Final Rearrangement

- The codewords for the most improbable and second-most improbable symbols must have the same length.
- The most improbable symbol's codeword also has a "sibling" of the same length.
- We can swap codewords to make this sibling be the codeword for the second-most improbable symbol. For the example, the result is:



An End and a Beginning

Shannon's Noiseless Coding Theorem is mathematically satisfying. From a practical point of view, though, we still have two problems:

- How can we compress data to nearly the entropy *in practice*?

  The number of possible blocks of size N is I^N — huge when N is large. And N sometimes must be large to get close to the entropy by encoding blocks of size N.

  One solution: A technique known as *arithmetic coding*.
- Where do the symbol probabilities $p_1, \ldots, p_I$ come from? And are symbols really independent, with known, constant probabilities?

  This is the problem of *source modeling*.