CSC 310: Information Theory

University of Toronto, Fall 2011

Instructor: Radford M. Neal

Week 4

Extensions of a Source

We formalize the notion of encoding symbols in blocks by defining the N-th *extension* of a source, in which we look at sequences of symbols, written as $(X_1, \ldots, X_N)$ or X^N.

If our original source alphabet, $\mathcal{A}_X$, has I symbols, the source alphabet for its N-th extension, $\mathcal{A}_X^N$, will have I^N symbols — all possible blocks of N symbols from $\mathcal{A}_X$.

If the probabilities for symbols in $\mathcal{A}_X$ are $p_1, \ldots, p_I$, the probabilities for symbols in $\mathcal{A}_X^N$ are found by multiplying the p_i for all the symbols in the block. (This is appropriate when symbols are independent.)

For instance, if $N = 3$:

$$P((X_1, X_2, X_3) = (a_i, a_j, a_k)) \;=\; p_i \, p_j \, p_k$$

Entropy of an Extension

We now prove that $H(X^N) = NH(X)$:

$$
\begin{aligned}
H(X^N) &= \sum_{i_1=1}^{I} \cdots \sum_{i_N=1}^{I} p_{i_1} \cdots p_{i_N} \log\left(\frac{1}{p_{i_1} \cdots p_{i_N}}\right) \\
&= \sum_{i_1=1}^{I} \cdots \sum_{i_N=1}^{I} p_{i_1} \cdots p_{i_N} \sum_{j=1}^{N} \log\left(\frac{1}{p_{i_j}}\right) \\
&= \sum_{j=1}^{N} \sum_{i_1=1}^{I} \cdots \sum_{i_N=1}^{I} p_{i_1} \cdots p_{i_N} \log\left(\frac{1}{p_{i_j}}\right) \\
&= \sum_{j=1}^{N} \sum_{i_j=1}^{I} \sum_{i_k \text{ for } k \neq j} p_{i_1} \cdots p_{i_N} \log\left(\frac{1}{p_{i_j}}\right) \\
&= \sum_{j=1}^{N} \sum_{i_j=1}^{I} p_{i_j} \log\left(\frac{1}{p_{i_j}}\right) \times \sum_{i_k \text{ for } k \neq j} p_{i_1} \cdots p_{i_{j-1}} p_{i_{j+1}} \cdots p_{i_N} \\
&= \sum_{j=1}^{N} \sum_{i_j=1}^{I} p_{i_j} \log\left(\frac{1}{p_{i_j}}\right) = NH(X)
\end{aligned}
$$

Or just use the fact that $E(U + V) = E(U) + E(V)$.

Shannon's Noiseless Coding Theorem

Using extensions 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 whose expected codeword length is less than NR.

Proof of Shannon's Noiseless Coding Theorem

Consider coding the N-th extension of a source whose symbols have probabilities $p_1, \ldots, p_I$, using a binary Shannon-Fano code.

The Shannon-Fano code for blocks of N symbols will have expected codeword length, L_N, less than $1 + H(X^N) = 1 + NH(X)$.

The expected codeword length per original source symbol will therefore be less 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.

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. Running the Huffman procedure for this size alphabet (or just storing the resulting code) may be infeasible.

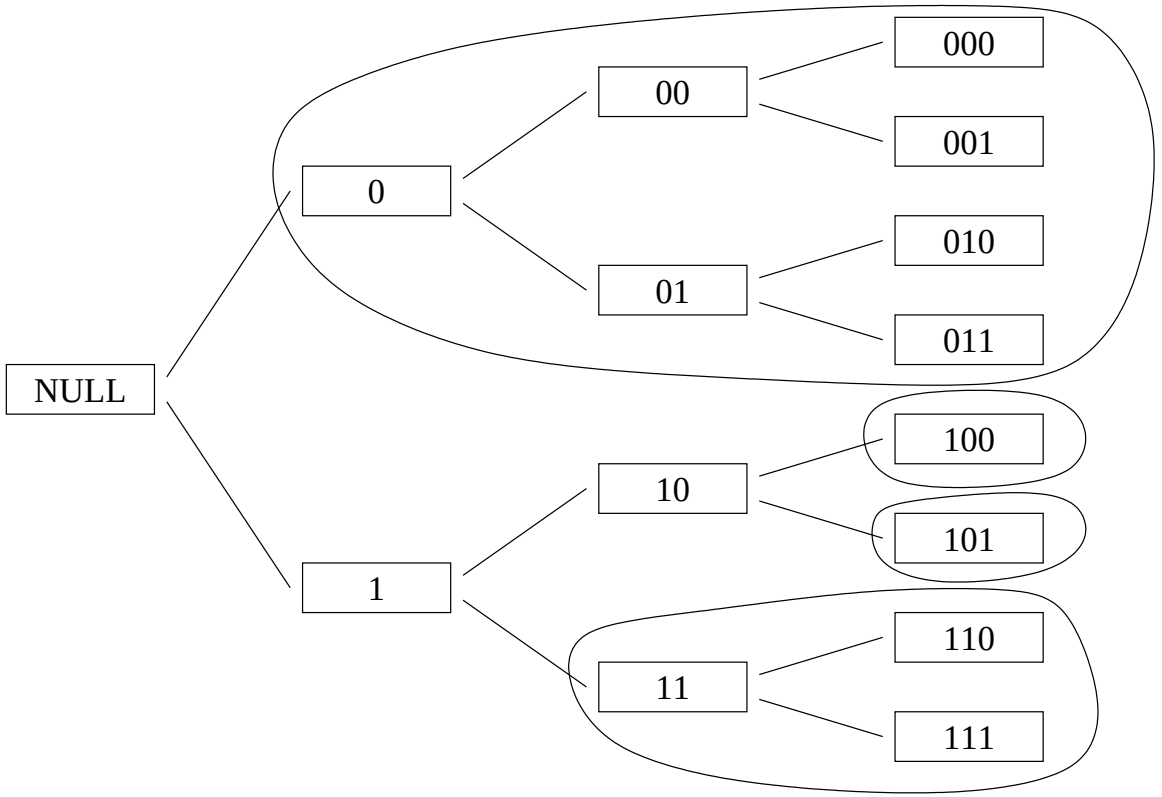
  One solution: A technique known as *arithmetic coding*.

- 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*.

Another Look at Code Trees

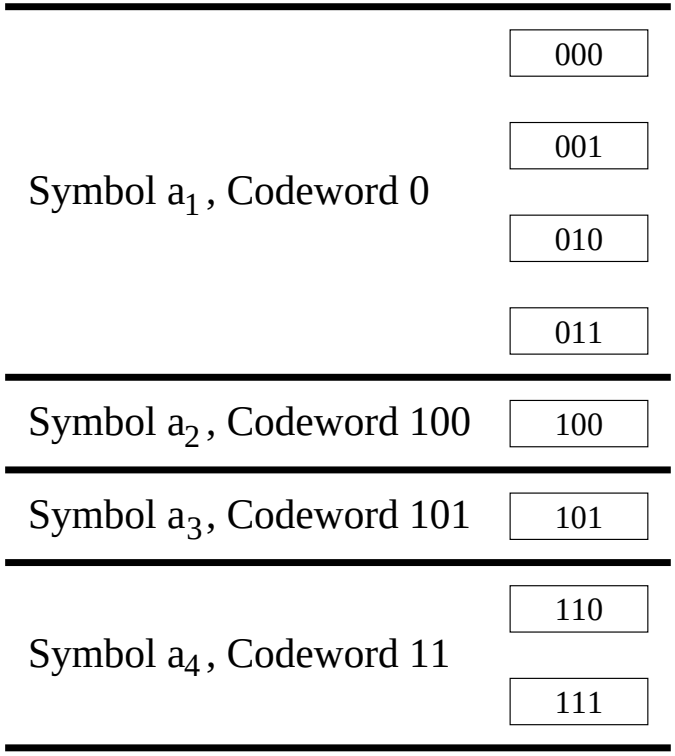
Any instantaneous code can be represented by a tree such as the following, with subtrees for codewords circled:



Rather than concentrate on the codewords that head each subtree, let's concentrate on the rightmost column...

Viewing a Code as a Way of Dividing up a "Codespace"

Here's the right column from the code tree, divided up according to codeword:

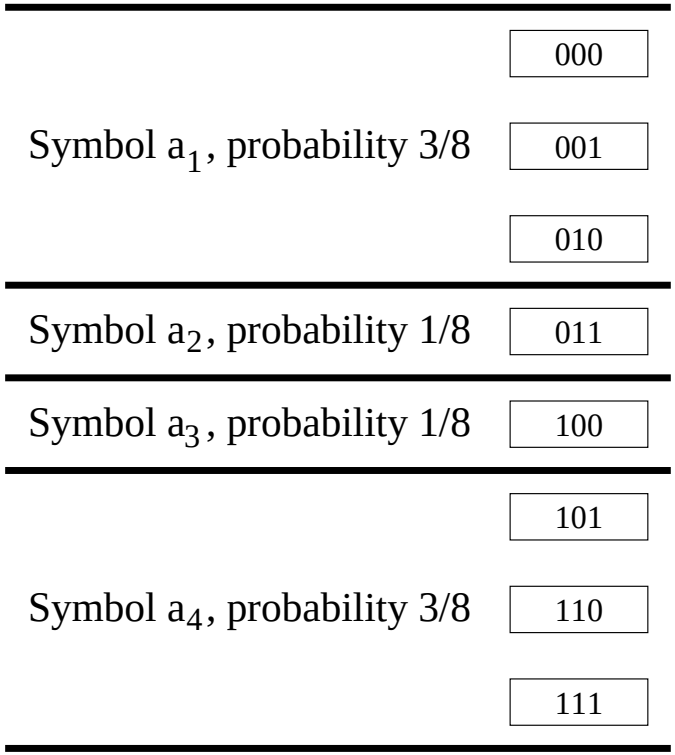


If we view $\{000, 001, 010, 011, 100, 101, 110, 111\}$ as an available "codespace", we see that this code divides it up so that symbol a_1 gets $1/2$ of it, symbols a_2 and a_3 get $1/8$, and symbol a_4 gets $1/4$.

Can We Use Other Divisions?

We know that this code is optimal if the fraction of codespace assigned to a symbol is equal to the symbol's probability.

But suppose the symbol probabilities were $3/8, 1/8, 1/8, 3/8$. We would then like to divide up codespace as follows:



Unfortunately, these divisions don't correspond to subtrees — so there's no code like this.

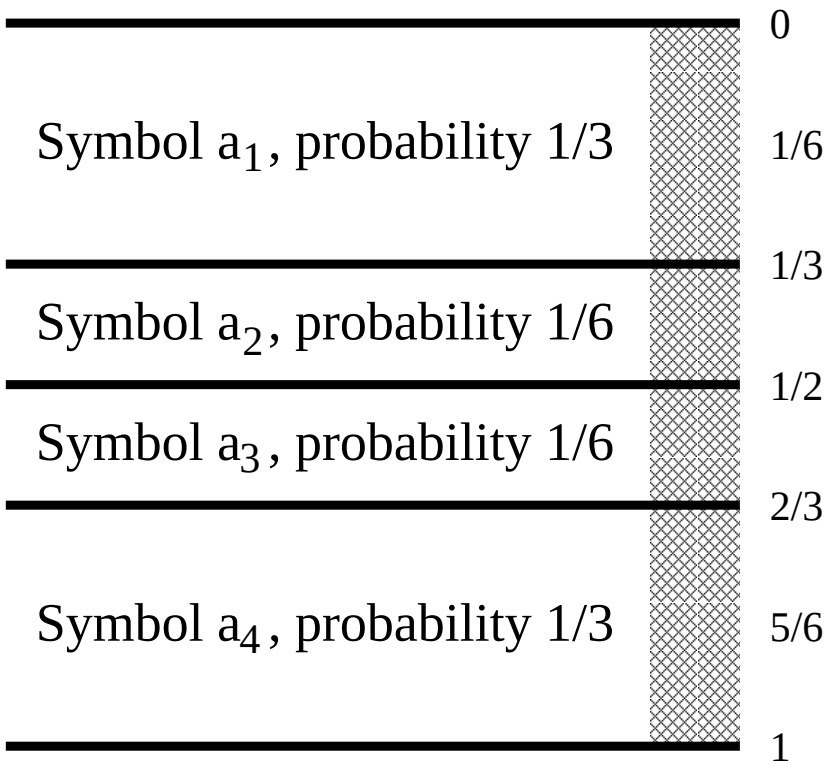
Viewing the Codespace as the Interval From 0 to 1

Let's ignore this problem of how to generate codewords for the moment.

Instead, let's ask how we could handle symbols that have probabilities like 1/3, which aren't multiples of 1/8.

A solution: Consider the codespace to be the interval of real numbers between 0 and 1.

For example:

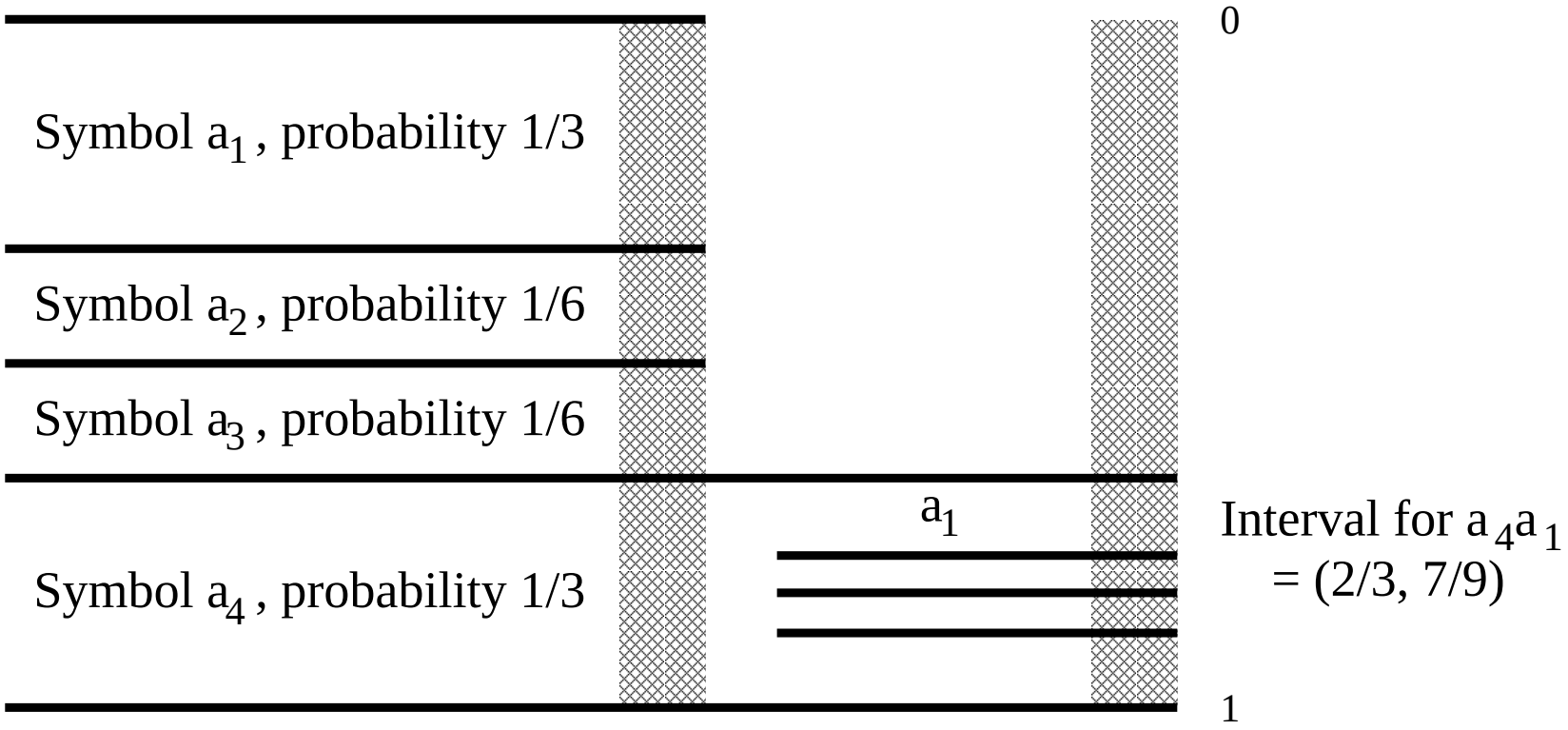


A Key Idea: Encode Blocks by Subdividing Further

Suppose we want to encode blocks of two symbols from this source.

We can do this by just subdividing the interval corresponding to the first symbol in the block, in the same way we subdivided the original interval.

Here's, how we encode the block $a_4\, a_1$:



Encoding Large Blocks as Intervals

Here's a general scheme for encoding a block of N symbols, $a_{i_1}, \ldots, a_{i_N}$:

1) Initialize the interval to $\left[u^{(0)},\ v^{(0)}\right)$, where $u^{(0)} = 0$ and $v^{(0)} = 1$.

2) For $k = 1, \ldots, N$:

   Let $u^{(k)} = u^{(k-1)} + \left(v^{(k-1)} - u^{(k-1)}\right) \sum_{j=1}^{i_k - 1} p_j$

   Let $v^{(k)} = u^{(k)} + \left(v^{(k-1)} - u^{(k-1)}\right) p_{i_k}$

3) Output a codeword that corresponds (somehow) to the final interval, $\left[u^{(N)},\ v^{(N)}\right)$.

This scheme is known as *arithmetic coding*, since codewords are found using arithmetic operations on the probabilities.

Finding a Codeword for an Interval

The last step requires that we be able to find a codeword for the final interval. We'll insist on an instantaneous code, for which no codeword is a prefix of another codeword.

Any binary codeword defines a number in $[0, 1)$, found by putting a "binary point" at its left end. Eg, the codeword 101 defines the number $1 \times (1/2) + 0 \times (1/4) + 1 \times (1/8)$.

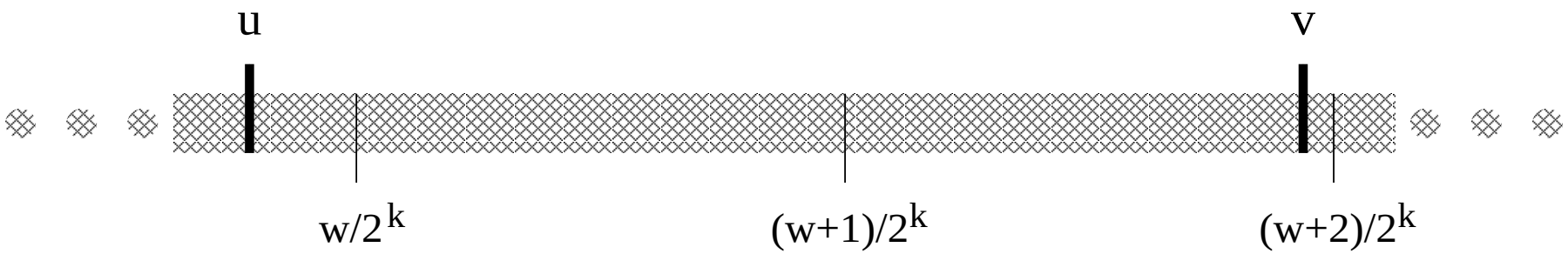
We'll choose a codeword such that:

- The codeword defines a point in the final interval.
- If we added any string of bits to the end of the codeword, it would still define a point in the final interval.

Codewords chosen in this way will form a prefix code for the blocks.

How Long Will the Codewords Be?

Here's a picture of how we pick a codeword for an interval:



Here, the interval $[w/2^k, (w+1)/2^k)$ fits entirely within $[u, v)$, the final interval found when encoding the block. We can therefore use the k-bit binary representation of w as the codeword for this block.

This can only be true if $v-u \geq 1/2^k$. Also, we will always be able to find such a codeword of length k if $v-u \geq 2/2^k = 1/2^{k-1}$ (as above).

Conclusion: We can pick a codeword of length k for a block of probability p $(= v-u)$ if $k \geq \log(1/p) + 1$. So codewords need be no longer than $\lceil \log(1/p) \rceil + 1$.

Getting Close to the Entropy Using Arithmetic Coding

We encode symbols from $\mathcal{A}_X$ in blocks of size N (ie, we use the N-th extension, $\mathcal{A}_X^N$), with N being quite large.

Assuming independence, the probability of the block $a_{i_1}, \ldots, a_{i_N}$ is $p_b = p_{i_1} \cdots p_{i_N}$.

We can find the interval for this block by subdividing $(0, 1)$ N times — *without* explicitly considering all possible blocks.

We can then find a binary codeword for this block that is no longer than

$$\lceil \log(1/p_b) \rceil + 1 < \log(1/p_b) + 2$$

The average codeword length for blocks will be less than

$$2 + \sum_b p_b \log(1/p_b) \;=\; 2 + H(\mathcal{A}_X^N) \;=\; 2 + NH(\mathcal{A}_X)$$

The average number of bits transmitted per symbol of $\mathcal{A}_X$ will be less than $H(\mathcal{A}_X) + 2/N$.

How Well it Works (So Far)

Big advantage:

We can get arbitrarily close to the entropy using big blocks, without an exponential growth in complexity with block size.

Big disadvantage:

If we use big blocks, many block probabilities will be tiny. For the procedure to work, we will have to use highly precise arithmetic.

(The number of bits of precision needed for a good approximation will go up linearly with blocksize, and the time for arithmetic involving such operands will also grow linearly.)

Fortunately, this disadvantage can be overcome.