3. Arithmetic coding / decoding
10-04, 10-08, 10-11  SXD

3.1. Segment coding
3.2. Binary representation of segments
3.3. Arithmetic coding
3.4. Effective computations
3.5. Arithmetic decoding
3.1. Segment coding

Consider a source alphabet $\mathcal{A} = \{a_1, ..., a_n\}$, with probabilities $\{p_1, ..., p_n\}$. Let $M = a_{j_1} \cdots a_{j_N}$ be a source message of length $N$. In segment coding (the first step in what will be arithmetic coding) a segment (or interval)

$$S_M = [l_M, h_M) \subset [0,1)$$

is assigned to any such $M$. Moreover, this assignment has the following properties:

1. $h_M - l_M = P(M)$
2. $S_M \cap S_{M'} = \emptyset$ for any pair two distinct messages $M, M'$ of length $N$.
3. $\bigcup_M S_M = [0,1)$.
4. If $M$ is a prefix of $M'$, then $S_{M'} \subset S_M$.

So the length of $S_M$ is $P(M)$; the different $S_M$, for fixed $N$, form a partition of $[0,1)$, and the partition for $N' > N$ is a refinement of that for $N$. 
The case \( N = 1 \)

For messages of length 1 (any one of the symbols \( a_j \) of \( \mathcal{A} \)), we set

\[
S_{a_j} = [p_1 + \cdots + p_{j-1}, p_1 + \cdots + p_{j-1} + p_j) = [\sigma_{j-1}, \sigma_j),
\]

where we define \( \sigma_j = p_1 + \cdots + p_{j-1} + p_j, j = 1, \ldots, n \) (we say that \( \sigma_1, \ldots, \sigma_n \) is the cumulative probability distribution. Thus we have

\[
l_{a_j} = \sigma_{j-1}, \quad h_{a_j} = \sigma_j, \quad h_{a_j} - l_{a_j} = p_j
\]

From the definitions it follows that \( 0 < p_1 = \sigma_1 < \cdots < \sigma_n = 1 \) and hence that the segments \( S_{a_j} \) cover \([0,1)\) and are pairwise disjoint. See the left column of the illustration on next page for an example in which \( n = 3 \).

Example. Before considering the general construction, we first describe it in a simple example. We will see how to obtain the segment \( S_M \) for the message \( babc \) produced by the source \( \{ 'a': 0.2, 'b': 0.5, 'c': 0.3 \} \).
By our stipulation of the case \( N = 1 \), we know that
\[
S_b = [0.2, 0.7).
\]
Let us proceed now to assign an interval to \( ba \) (second column of the illustration), then to \( bab \) (third column) and finally to \( babc \) (fourth column). The interval \( S_{ba} \) is obtained by subdividing
\[
S_b = [l_b = 0.2, h_b = 0.7)
\]
into segments of relative length \( p(a) = 0.2 \), \( p(b) = 0.5 \) and \( p(c) = 0.3 \) and choosing the \( a \)-segment as \( S_{ba} \). So the division points are
\[
l_{ba} = 0.2, \quad h_{ba} = l_{bb} = 0.2 + 0.5 \times \sigma(a) = 0.30,
\]
\[
h_{bb} = l_{bc} = 0.2 + 0.5 \times \sigma(b) = 0.55, \quad h_{bc} = 0.2 + 0.5 \times \sigma(c) = 0.70, \text{ so}
\]
\[
S_{ba} = [0.2, 0.3), \quad S_{bb} = [0.30, 0.55), \quad S_{bc} = [0.55, 0.70).
\]
Now the interval $S_{bab}$ is obtained in a similar way: subdivide $S_{ba}$ into intervals that are proportional to $p(a)$, $p(b)$ and $p(c)$ and choose as $S_{bab}$ the segment corresponding to $b$. Actually we have

$$h_{baa} = l_{bab} = 0.2 + 0.1 \times \sigma(a) = 0.22,$$

$$h_{bab} = l_{bac} = 0.2 + 0.1 \times \sigma(b) = 0.27,$$

$$h_{bac} = 0.2 + 0.1 \times \sigma(c) = 0.30,$$

and hence

$$S_{bab} = [l_{bab}, h_{bab}) = [0.22,0.27).$$

Finally we get, following the same procedure with $S_{bab}$,

$$S_{babc} = [0.22 + 0.05 \times \sigma(b), 0.22 + 0.05 \times \sigma(c)) = [0.255,0.270).$$
**The general case**

Suppose that we already know the interval $S_M = [l_M, h_M)$ of a message of length $N$ and that $h_M - l_M = P(M)$. Then the interval of $M' = Ma_j$ is defined as follows:

$$S_{M'} = [l_{M'}, h_{M'}) = [l_M + P(M) \times \sigma_{j-1}, l_M + P(M) \times \sigma_j).$$

We note that

$$h_{M'} - l_{M'} = P(M) \times (\sigma_j - \sigma_{j-1}) = P(M)p_j = P(M').$$

**Remark.** The conditions 1-4 at the beginning are a direct consequence of the definitions.
3.2. Binary representation of segments

**Binary representation of numbers in the unit segment**

The binary unit segment

**Examples.** 0.5→0.1, 0.25→0.01, 0.75→0.11, 0.125→0.001;
0.255→0.01000, 0.270→0.01001;
0.011001→1/4+1/8+1/64=25/64=0.390625.
3.3. Arithmetic coding

The basic idea of *arithmetic coding* is to select an element in the set \( S_M = [l_M, h_M] \) that requires the minimal number of bits. Then we encode \( M \) using the binary word formed with those bits.

This can be accomplished as follows. Suppose that the first bit that is different in the binary representations of \( l_M \) and \( h_M \) is the \( r \)-th, so that we will have

\[
l_M = 0.b_1b_2 \cdots b_{r-1}0 \cdots, \quad h_M = 0.b_1b_2 \cdots b_{r-1}1 \cdots
\]

If \( 0.b_1b_2 \cdots b_{r-1}1 < h_M \), then the number \( 0.b_1b_2 \cdots b_{r-1}1 \), or the word \( b_1b_2 \cdots b_{r-1}1 \), satisfies the requirements, for \( 0.b_1b_2 \cdots b_{r-1}1 \in S_M \) and any other number in the interval \( S_M \) will require more bits. Thus we encode \( M \) as \( b_1b_2 \cdots b_{r-1}1 \). If \( 0.b_1b_2 \cdots b_{r-1}1 = h_M \), then the shortest binary word will be \( b_1b_2 \cdots b_{r-1}1 \) if \( l_M = 0.b_1b_2 \cdots b_{r-1}0 \). Otherwise \( l_M = 0.b_1b_2 \cdots b_{r-1}0x \), with \( x \neq 0 \). If \( x = 0 \cdots \), we can take \( 0.b_1b_2 \cdots b_{r-1}01 \), else if \( x = 1 \cdots 10 \) (\( s \) ones), we may take \( 0.b_1b_2 \cdots b_{r-1}01 \cdots 11 \) (\( s + 1 \) ones).
**Remark.** For the decoding, it will be convenient to include in the encoding the length of the original message. Thus it can be represented as a pair \((L, c)\), where \(L\) is the length of the message and \(c\) is the binary string given by the arithmetic encoding.

For example, if \(S = [\text{‘a’ : 0.2, ‘b’ : 0.5: ‘c’ : 0.3}]\) and \(M=\text{”babc”}\), then the arithmetic encoding is \([4,”010001”]\), for the decimals of 0.255 and 0.270 are, respectively:

01000001010 ⋯

01000101000 ⋯
3.4. Effective computations

def accumulate(P):
    S=[]       # for the accumulated distribution
    a=0.0      # the current accumulated value
    for x in P:
        a += x[1]   # x[1]: probability of s=x[0]
        S.append((x[0],a))   # add pair (s,a) to S.
    return(S)

def IE(M, P):
    S=accumulate(P)
    P=dict(P); S=dict(S)
    l=0.0; h=1.0; u=h-l
    for x in M:   # see formulas on page 6
        h=l+S[x]*u; u=P[x]*u; l=h-u
    return [l,h]
def BE(a,b,nb=58):
    l=dec2bin(a,nb)
    h=dec2bin(b,nb)
    r=0
    while l[r]==h[r]:  # see discussion of page 8
        r=r+1
        h=h[:r]+'1'
    if bin2dec(h)<b: return h
    else:
        if bin2dec(l[:r])==a: return l[:r]
        x=l[(r+1):]
        if x[0]=='0': return l[:r]+'01'
        j=x.index('0')  # it will fail if no 0 in x
        return l[:r+1+j]+'1'

def AE(M, P, nb=58):
    l, h = IE(M, P)
    return([len(M),BE(l,h,nb)])
3.5. Arithmetic decoding

Suppose we have the arithmetically encoded message

\[ C = [N, x] \]  \( (N \) the number of symbols, \( x \) the binary string encoding). \]

Let \( P \) be a representation of the source as a list of pairs (symbol, probability). Let \( A \) denote the list of pairs (symbol, cumulative-probability). The decoding algorithm can be described as follows:

```python
def AD(C,P):
    N = C[0]; x=C[1]; x=bin2dec(x)
    A=accumulate(P); P=dict(P)
    l=0; h=1; M=""
    for j in range(N):
        u = h-l
        for (k,q) in A:
            if (l+q*u <= x): continue
            else: break
        M += k; h = l + q*u; l = h-P[k]*u
    return M
```
**Example.** If \( P=\{(\text{'a'}, 0.25), (\text{'b'}, 0.4), (\text{'c'}, 0.15), (\text{'d'}, 0.1), (\text{'e'}, 0.1)\} \) and 
\( M=\text{"badbbdcbabea"}, \)
then the arithmetic encoding \( \text{AE}(M,P) \) is 
\[
[12,010101011011101101110010110100111]
\]

(see the examples in AE-AD.py).

**Remark.** This description of arithmetic encoding and decoding would work if floats had unlimited precision, but in pyzo it only works for short messages (of the order of 54 encoded bits, due to the fact that floats have 16 significant digits, which amounts to \( 16\log(10)/\log(2) \approx 53.15 \) bits.)