

Cesáro Mean Example

Theorem 4.2.3 on page 64 says: If $\lim_{n \rightarrow \infty} a_n = a$ and $b_n = \frac{1}{n} \sum_{i=1}^n a_i$, then $\lim_{n \rightarrow \infty} b_n = a$.

Let:

$$\begin{aligned} a_n &= 1 + \frac{1}{n} \\ b_n &= \frac{1}{n} \sum_{i=1}^n a_i \\ c_n &= \sum_{i=1}^n a_i \end{aligned}$$

Notice that $b_n = 1 + \frac{1}{n} \sum_{i=1}^n \frac{1}{i}$ so $1 < b_n < 1 + \frac{1 + \log(n)}{n}$ and $\lim_{n \rightarrow \infty} b_n = 1$ as the theorem promises. Figure 1 shows a_n, b_n , and c_n for $n \in [1, 2, \dots, 10]$. The slope of a plot of c at n is a_n and it converges, $\lim_{n \rightarrow \infty} (c_n - c_{n-1}) = 1$, but there is no straight line that a plot of c approaches. Define y_n to be the intercept of the line through c_n and c_{n-1} .

$$\begin{aligned} y_n &= c_n - n(c_n - c_{n-1}) \\ &= \sum_{i=1}^n 1 + \frac{1}{i} - n \left(1 + \frac{1}{n} \right) \\ &= n + \sum_{i=1}^n \frac{1}{i} - n \left(1 + \frac{1}{n} \right) \\ &= \sum_{i=2}^n \frac{1}{i}. \end{aligned}$$

Informal presentation of Lagrange multipliers¹

Two scalar functions of 2-dimensional vectors appear in Fig. 2. Intuition (correctly) suggests that at the point x^* where the lower function f is minimized subject to a constraint, say $g(x) - 8 = 0$, on the upper function, that the level sets of g and f are tangent. Having the level sets of f and g

¹This material comes from my course notes for *Systems Science 512: Quantitative Methods of Systems Science*. The whole set is at both <ftp://ftp.ece.pdx.edu/pub/users/andy/sysc512/> and <http://www.ece.pdx.edu/~andy/512/>

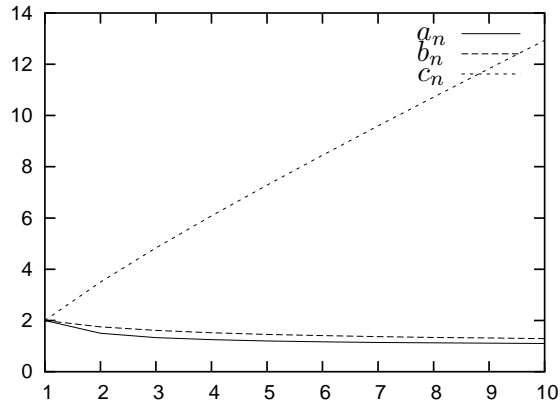


Figure 1: Plot of a_n , b_n , and c_n defined in the text. The point is that any line with a slope of 1 and an intercept greater than 2 will cross c_n no matter how large the intercept.

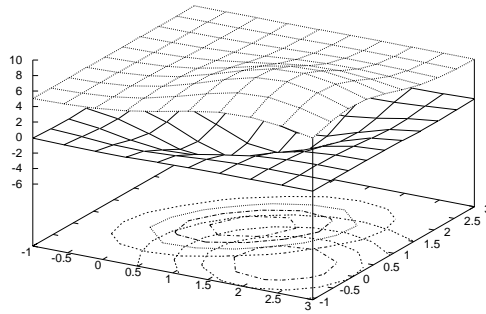


Figure 2: Constrained optimization. If the lower function f is to be minimized and the solution is required to lie on a level set of the upper function g , then at the solution the contour lines for the two functions will be tangent. The task is to find the vector x^* that *minimizes* $f(x)$ subject to the constraint $c(x) \equiv g(x) - 8 = 0$.

tangent at x^* is equivalent to the gradients being parallel:

$$\left. \frac{\partial}{\partial x_i} f(x) \right|_{x=x^*} = \lambda \left. \frac{\partial}{\partial x_i} c(x) \right|_{x=x^*} \quad i \in \{1, 2\}$$

By defining a *Lagrangian*

$$\mathcal{L}(x, \lambda) = f(x) - \lambda c(x)$$

the tangency requirement can be stated as:

$$\left. \frac{\partial}{\partial x_i} \mathcal{L}(x, \lambda) \right|_{x=x^*, \lambda=\lambda^*} = 0. \quad (1)$$

The auxiliary variable λ is a *Lagrange multiplier*, and using Eqn. 1 to find x^* is called the Lagrange multiplier method.

That tangency of the level sets of f and c is required for either a minimum or a maximum, suggests that tangency alone is not sufficient to identify an optimum. The Lagrangian \mathcal{L} also appears in the second order conditions that are sufficient to identify a local optimum. These are more complicated than the first order tangency condition. After using Eqn. 1 to identify a possible optimum \bar{x} , it may be possible to use some properties of the problem to verify that \bar{x} is indeed an optimum without going to the trouble of checking the second order conditions.

Costs of Blocking

See Ross Williams' dissertation (<http://www.ross.net/compression/index.html>) for an informative rant on the weaknesses of Huffman coding. Huffman coding is optimal for single letters (Williams calls these *instances*). The problem is that you almost always want to send more than a single letter. While in principal you can build Huffman codes for however big a thing you want to send, it is not practically possible. The usual practice is to break a message into blocks of letters and code the blocks separately.

There are two costs of blocking:

- Modeling cost (see Fig. 3).
- Coding cost (can be limited to one bit per block by Huffman coding or two bits per block by Arithmetic coding).

Williams targeted these two costs on page 18 of his dissertation, writing:

[...] two mistaken assumptions [...]

The first was that instances in a message are independent of one another.

The second was that each source symbol must be mapped onto a discrete number of channel symbols.

Given $P_{X^{(t)}|X_1^{t-1}}$, i.e., a *model*, one can use arithmetic coding to represent an entire message in a single block, thus paying the blocking costs only once. One way to view this is that the number of bits used to represent a letter is allowed to be a non-integer rational number.

Suppose that we use blocks of size n and use the following notation:

P_{X^*} The *true* model.

$Q_{X_1^n}$ The best block model.

$R_{X_1^n}$ The best dyadic block model, ie, $R(x_1^n) = 2^{-l(x_1^n)}$ where

$$l(x_1^n)$$

is the length of the codeword assigned to x_1^n by a Huffman code.

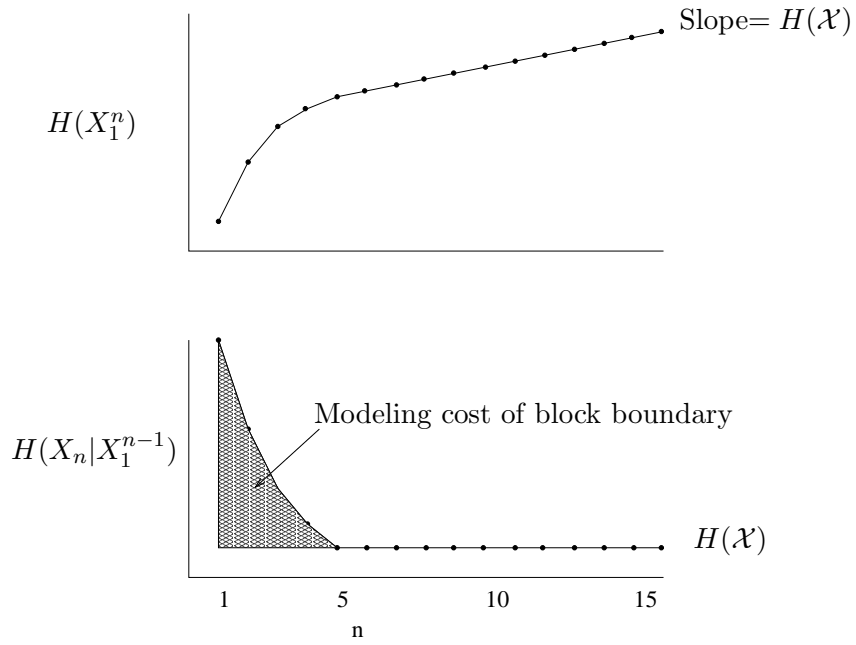


Figure 3: Modeling cost of blocking. The upper plot illustrates the definition of entropy rate, ie, $H(\mathcal{X}) = \lim_{n \rightarrow \infty} \frac{1}{n} H(X_1^n)$. The lower plot illustrates the cost of disregarding the history at block boundaries.

Asymptotically, the cost of blocking on a per symbol basis is given by the relative entropy rate²,

$$\begin{aligned} d(P||R) &\equiv \frac{1}{n} E_P \log \frac{P(X_1^n | X_{-\infty}^0)}{R(X_1^n)} \\ &= d(P||Q) + d(Q||R) \end{aligned} \quad (2)$$

as illustrated in Fig. 4.

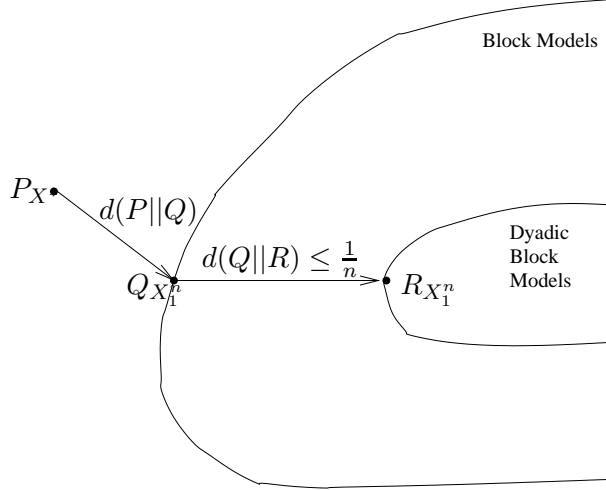


Figure 4: There are two parts of the cost of blocking. The modeling cost comes from the constraint that the probability function Q lie in the class of *Block Models*. The coding cost comes from the additional constraint that the probability function R lie in the class of *Dyadic Block Models*.

²The following shows d adds as in Eqn. (2):

$$\begin{aligned} nd(P||R) &= E_P \log \frac{P(X_1^n | X_{-\infty}^0)}{R(X_1^n)} \\ &= E_P \log \left(\frac{P(X_1^n | X_{-\infty}^0)}{Q(X_1^n)} \frac{Q(X_1^n)}{R(X_1^n)} \right) \\ &= nd(P||Q) + E_P \log \left(\frac{Q(X_1^n)}{R(X_1^n)} \right) \\ &= nd(P||Q) + E_Q \log \left(\frac{Q(X_1^n)}{R(X_1^n)} \right) \\ &= nd(P||Q) + nd(Q||R) \end{aligned}$$

Compression as a Change of Variable

Consider the problem of encoding strings of *trits* as strings of bits. Let x_1^n , with $x_i \in \{0, 1, 2\}$, represent a string of trits and b_1^m represent a string of bits. Such strings can be mapped to points in the interval $[0, 1)$ by:

$$x_1^n \leftrightarrow \sum_{t=1}^n \frac{x_t}{3^t}$$
$$b_1^m \leftrightarrow \sum_{t=1}^m \frac{b_t}{2^t}.$$

If the X 's are i.i.d., the distribution of X_1^n in the limit $n \rightarrow \infty$ produces a fractal measure. The particular case $\Pr(X = 0) = \Pr(X = 2) = \frac{1}{2}$ and $\Pr(X = 1) = 0$ yields the classic middle third Cantor set. I've illustrated the case $\Pr(X = 0) = \Pr(X = 2) = \frac{2}{5}$ and $\Pr(X = 1) = \frac{1}{5}$ in Fig. 5 On the other hand, for a good code, the b 's will be i.i.d. with $\Pr(B = 0) = \Pr(B = 1) = \frac{1}{2}$. Thus a good code is a transformation that changes a fractal measure into a uniform measure.

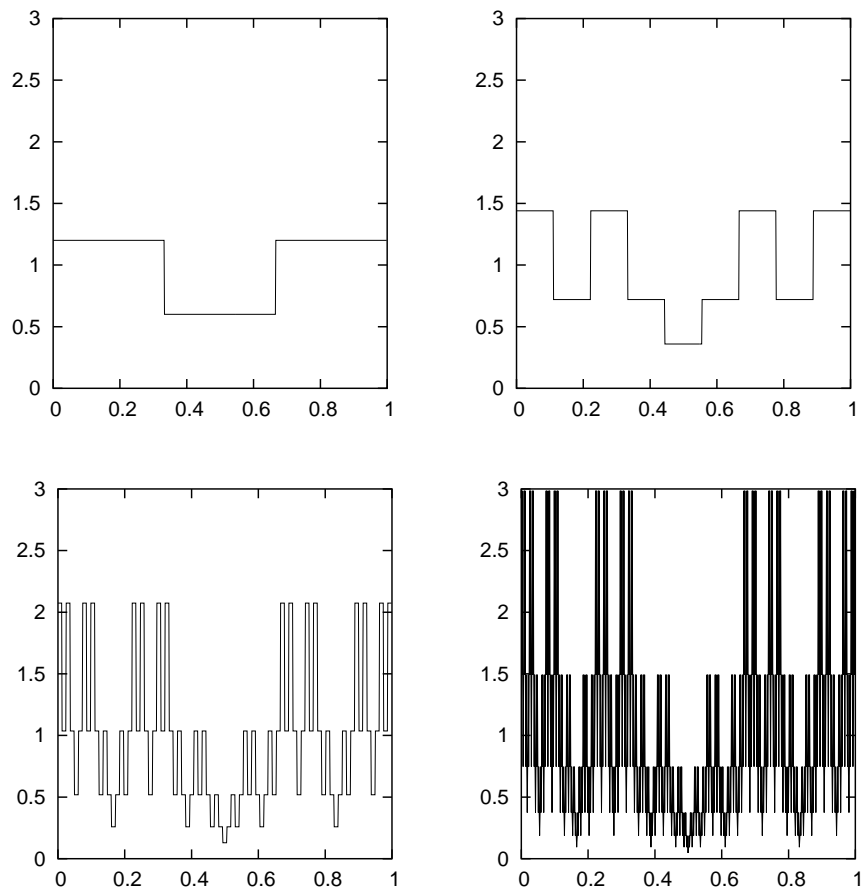


Figure 5: Plots of approximations of a fractal density. I made the plots by assigning the following probabilities to *trits* $\{\Pr(0) = \frac{2}{5}, \Pr(1) = \frac{1}{5}, \Pr(2) = \frac{2}{5}\}$ and considering base three representations of points in the interval $[0, 1)$.

Notes on arithmetic coding

In class I will go through the rescaling that Witten Neal and Cleary described in Fig. 1b of their 1987 paper *Arithmetic Coding for Data Compression* in the Communications of the ACM. I found a copy of the paper at <http://portal.acm.org/citation.cfm?doid=214762.214771>

The python program *ac1.py* reproduces the scalings of Fig. 1b of the Witten Neal Cleary paper.

```
import sys

# Initialize message to the indices for 'eaii!'
message = [1,0,2,2,5]
# Initialize the probabilities for the alphabet
prob = [.2, .3, .1, .2, .1, .1]
# Initialize the working interval
interval = [0.0,1.0]

# The main loop builds up a table of ranges on the interval.
# table[t] consists of the decision points for possible values
# of message[t+1] given that the values of message[0] ... message[t]
# have been seen.
table = []
for x in message:
    divisions = []
    text = []
    c = interval[0]
    l = interval[1] - interval[0]
    divisions.append(c)
    text.append("%7.5f"%c)
    for p in prob:
        c = c + p*l
        divisions.append(c)
        text.append("%7.5f"%c)
    table.append(text)
    interval[0] = divisions[x]
    interval[1] = divisions[x+1]

# This loop prints out the table in a format suitable for latex
print "
beginverbatim"
```

```

for y in range(0,len(divisions)):
    for x in range(0,len(table)):
        sys.stdout.write(table[x][y]+" ")
    print " "
print "
endverbatim"

```

Here is the output of the `ac1.py`

```

0.00000 0.20000 0.20000 0.23000 0.23300
0.20000 0.26000 0.21200 0.23120 0.23312
0.50000 0.35000 0.23000 0.23300 0.23330
0.60000 0.38000 0.23600 0.23360 0.23336
0.80000 0.44000 0.24800 0.23480 0.23348
0.90000 0.47000 0.25400 0.23540 0.23354
1.00000 0.50000 0.26000 0.23600 0.23360

```

Program `ac2.py` assumes the same probabilities and processes the same input sequence, but, like an arithmetic coder, it expands the scale by a factor of 2 when ever possible. Here is the python program `ac2.py`.

```

import sys

# Initialize message to the indices for 'eaii!'
message = [1,0,2,2,5]
# Initialize the probabilities for the alphabet
prob = [.2, .3, .1, .2, .1, .1]
# Initialize the working interval
interval = [0.0,1.0]

# The main loop builds up a table of ranges on the interval. table[t]
# consists of the decision points for possible values of message[t+1]
# given that the values of message[0] ... message[t] have been seen.
# The interval is rescaled whenever it does not include 0.5 or its
# length is less than 0.25

table = []
for x in message:
    divisions = []
    text = []

```

```

while ((interval[1] < 0.5) | (interval[0] > 0.5) | (interval[1] - interval[0] < 0.25))
  if interval[1] < 0.5:
    interval[0] = interval[0] * 2
    interval[1] = interval[1] * 2
  else:
    if interval[0] > 0.5:
      interval[0] = interval[0] * 2 - 1
      interval[1] = interval[1] * 2 - 1
    else:
      if interval[1] - interval[0] < 0.25:
        interval[0] = interval[0] * 2 - 0.5
        interval[1] = interval[1] * 2 - 0.5
  c = interval[0]
  l = interval[1] - interval[0]
  divisions.append(c)
  text.append("%7.5f"%c)
  for p in prob:
    c = c + p*l
    divisions.append(c)
    text.append("%7.5f"%c)
  table.append(text)
  interval[0] = divisions[x]
  interval[1] = divisions[x+1]

# This loop prints out the table in a format suitable for latex
print "
beginverbatim"
for y in range(0,len(divisions)):
  for x in range(0,len(table)):
    sys.stdout.write(table[x][y]+" ")
  print " "
print "
endverbatim"

```

Here is the output of the python program *ac2.py*.

```

0.00000 0.20000 0.10000 0.22000 0.29600
0.20000 0.26000 0.19600 0.29680 0.35744
0.50000 0.35000 0.34000 0.41200 0.44960
0.60000 0.38000 0.38800 0.45040 0.48032

```

0.80000 0.44000 0.48400 0.52720 0.54176
0.90000 0.47000 0.53200 0.56560 0.57248
1.00000 0.50000 0.58000 0.60400 0.60320