Outstanding Reading

- Read chapters 1, and 2 in C&T.
- Read chapter 3 in C&T.
- Read section 11.1,11.3, method of types and universal source coding.
- Read chapter 4.
- Read chapter 5.
- Read stream code chapter 6 in “Information Theory, Inference, and Learning Algorithms” by David J.C. MacKay (available online http://www.inference.phy.cam.ac.uk/mackay/itila/)
- Read chapter 7 in Cover and Thomas, channel capacity
Announcements, Assignments, and Reminders

- Next homework won’t go out until next week (no homework over the weekend! 😊)
- Late policy: 10% every 24 hour period that you are late, and no more than 3 days late accepted.
- Lowest grade out of all HW grades is not counted towards final grade (so you can skip one HW with impunity).
- Please do use our discussion board (https://catalyst.uw.edu/gopost/board/karna/25503/) for all questions, so that all will benefit from them being answered.

Class Road Map

- L1 (1/3): Overview, Entropy
- L2 (1/5): Props. Entropy, Mutual Information, KL-Divergence
- L3 (1/10): KL-Divergence, Jensen, properties, Data Proc. Inequality
- L5 (1/17): Fano, AEP
- L6 (1/19): snow
- L6 (1/24): AEP, source coding
- L7 (1/26): Method of Types
- L9 (2/2): HMMs, coding
- L10 (2/7): Coding, Kraft,
- L11 (2/9): Huffman, midterm
- L12 (2/14): **Midterm**
- L13 (2/16): Shannon Games, Arithmetic
- L14 (2/21): Channel Capacity
- L16 (2/28): Shannon’s 2nd theorem.
- L17 (3/1):
- L18 (3/6):
- L19 (3/8):

Finals Week: March 12th–16th.
Huffman Codes

- Huffman coding is a symbol code, we code one symbol at a time.
- Is Huffman optimal? But what does optimal mean?
- In general, for a symbol code, each symbol in the source alphabet must use an integer number of codeword bits.
- This is ok for $D$-adic distributions but could use up to one extra bit per symbol on average.
- Bad example: $p(0) = 1 - p(1) = 0.999$, then $-\log p(0) \approx 0$, so we should be using close to zero bits per symbol to code this, but Huffman uses 1.
- Thus, we need a long block to get any benefit.
- In practice, this means we need to store and be able to compute $p(x_1:n)$. No problem, right?

Can we easily compute $p(x_1:n)$?

- If $|A|$ is the alphabet size, we need a table of size $|A|^n$ to store these probabilities.
- Moreover, it is hard to estimate $p(x_1:n)$ accurately. Given an amount of “training data” (to borrow a phrase from machine learning), it is hard to estimate this distribution. Many of the possible strings in any finite sample size will not occur (sparsity).
- Example: how hard is it to find a short valid English phrase never before written using a web search engine?
- Smoothing models are required. Similar to the language model problem in natural language processing.
Huffman Codes

- Huffman has the property that

\[ H(X) \leq L(\text{Huffman}) \leq H(X) + 1 \]  

(1)

- Bigger block sizes help, but we get

\[ H(X_{1:n}) \leq L(\text{Block Huffman}) \leq H(X_{1:n}) + 1 \]  

(2)

for the block.

- If \( H(X_{1:n}) \) is small (e.g., English text) then this extra bit can be significant.

- If block gets too long, we have the estimation problem again (hard to compute \( p(x_{1:n}) \),

- also the fact that it introduces latencies (we need to encode and then wait for the end of a block before we can send any bits).

The Probabilities They Are A-Changin’

- Real sequential processes are not stationary. It might be a reasonable approximation to assume that they are “locally stationary”, meaning that the statistics of the process are governed by a distribution \( p(x) \) within a given fixed-width time window.

- Huffman assumes one fixed \( p(x) \). If this changes, say to \( p'(x) \), the code will be less optimal by \( D(p'(x) || p(x)) \) bits per symbol, where \( p'(x) \) is the “correct” distribution.

- Instead we could:
  1. Recompute Huffman distribution and code each period. This is inefficient, however, as we’ll need to re-transmit the codebook each time!
  2. We could do some sort of adaptive Huffman scheme.

- But do we really want a Huffman code to begin with?
Consider English text. Redundancy abounds.
Redundancy exists at the sentence level, the word level, and the character level.
Complete the following sentence fragment: “with more than 300 dead, most of the victims choked to death.” did you really need to see that last word, we could just predict it, or alternatively use few bits to code it.
Shannon realized this early on, and he attempted to come up with an estimate of the entropy of English text.
We can use Humans and play a guessing game to help us do that.
Assume we have a simple alphabet, the letters 'A' through 'Z' along with space “ ” (so 27 symbols).
The process is as follows: Given a letter history, we ask a person to guess the following letter, and count how many guesses it takes to get it right. Letters are given one at a time in sequence.

• Sorry, you need to come to class to see this slide 😊
The sequence of guess numbers for the set of letters can be seen as a “code” for the string. I.e., a mapping from letters to integers:

\[ C : \{ 'A', 'B', 'C', \ldots, 'Z', ' ' \} \rightarrow \{ 1, 2, \ldots, 27 \} \] (3)

Often, the guesses are immediate. So there are many more ones (1s), twos (2s), and so on then there are large integers.

Things that are more predictable have fewer guesses, or have higher probability. Things that require more guesses are less predictable, and have lower probability.

The redundancy in English is its predictability, the more low numbered integers, the more English is redundant.

**Bits per guess**

- Let \( g_t \) be the number of guesses at position \( t \) within the source string.
- \( \log g_t \) is the number of bits to represent a number as large as \( g_t \), or number of bits required to encode number of guesses at stage \( t \).
- Then we can estimate the entropy rate of this process as follows:

\[ \hat{H}(X) \approx \frac{1}{n} \sum_{t=1}^{n} \log g_t \approx \frac{1}{n} \sum_{t=1}^{n} \log \frac{1}{p(x_t|x_{t-1}, x_{t-2}, \ldots, x_1)} \] (4)

Suppose that \( x_1, x_2, \ldots \) is a stochastic process with an entropy rate of the form \( H(X_t|X_{t-1}, \ldots, X_1) \). Then \( p(x_t|x_{t-1}, x_{t-2}, \ldots, x_1) \) is the probability of letter \( x_t \) at time \( t \). Then we should have approximately

\[ g_t \approx \frac{1}{p(x_t|x_{t-1}, x_{t-2}, \ldots, x_1)} \] (5)
Compression

- A compression algorithm could transform the source string into a string of numbers. I.e., we’d use $C : \{\text{A}', \text{B}', \text{C}', \ldots, \text{Z}', ' \} \rightarrow \{1, 2, \ldots, 27\}$ to transform from source symbols into code symbols.
- But the frequency of code symbols 1, 2, 3, etc. is much higher than any of the source (alpha) symbols. I.e., we’ll see a “1” much more frequently than an “e” since many letters, even if they are not an “e” are easily guessable, and sometimes “e” is not guessable.
- So, rather than encode “there is no reverse on a motorcycle”, we would encode and compress “1, 1, 1, 5, 1, 1, 2, 1, 1, 2, 1, 1, 15, 1, 1, 17, 1, 1, 1, 2, 1, 3, 2, 1, 2, 2, 7, 1, 1, 1, 4, 1, 1, 1, 1” should compress well, many 1s.

Decompression

- But how do we decode?
- We use an identical twin. I.e., at stage $t$, we ask that twin to guess the next letter, and tell them to stop guessing when they have made $g_t$ guesses.
- Thus we’ll recover the source message.
- Alternatively, we can compute a large table for all possible histories and memorize a Human’s guessing scheme. I.e., for all possible histories we would record $g_t$ and then wait that many guesses.
- This is of course impractical. $27^L$ for length $L$ strings, so we’ll need to do something smarter (as we will see).
- Nonetheless, this scheme, first introduced by Shannon in the early 1950s is the basis for what is called arithmetic coding.
Arithmetic Coding

- This is the method used by DjVu (adaptive image compression used for printed material, overtaken by PDF but probably certain PDF formats use this as well).
- Assume we are given a probabilistic model of the source. I.e.,

\[ p(x_{1:n}) = \prod_{i=1}^{n} p(x_i) \]  \hspace{1cm} (6)

or alternatively

\[ p(x_{1:n}) = p(x_1) \prod_{i=2}^{n} p(x_i | x_{i-1}) \]  \hspace{1cm} (7)

- Higher order Markov models often used as well (as we’ll see).

At each symbol, we use the conditional probability to provide the probability of the next symbol.

Arithmetic coding can easily handle complex adaptive models of the source that produce context-dependent predictive distributions (so not nec. stationary). E.g., could use \( p_t(x_t | x_1, \ldots, x_{t-1}) \)

Best understood with an example. Let \( \mathcal{X} = \{a, e, i, o, u, !\} \) so \( |\mathcal{X}| = 6 \).

Source \( X_1, X_2, \ldots \) need not be i.i.d.

Assume that \( p(x_n | x_1, x_2, \ldots, x_{n-1}) \) is given to both encoder (sender, compressor) and receiver (decoder, uncompressor).
Arithmetic Coding

- Like in Shannon-Fano-Elias coding, we divide the unit interval up into segments of length according to the probabilities \( p(X_1 = x) \) for \( x \in \{a, e, i, o, u!\} \).
- Consider the following figure:

Each subinterval may be further divided into segments of (relative) length \( p(X_2 = x_2 | X_1 = x_1) \) or actual length \( p(X_2 = x_2, X_1 = x_1) \).
- Relative lengths longer or shorter \( p(X_1 = j) \geq p(X_2 = j | X_1 = k) \).
- The following figure shows this, starting with \( p(X_1 = a) \).
Arithmetic Coding

- Length of interval for “ae” is
  \[ p(X_1 = a, X_2 = e) = p(X_1 = a)p(X_2 = e | X_1 = a) \]
- Intervals keep getting exponentially smaller with \( n \) larger.
- Key: at each stage, relative lengths of the intervals can change depending on history. At \( t = 1 \), relative interval fraction for “a” is \( p(a) \), at \( t = 2 \), relative interval fraction for “a” is \( p(a | X_1) \), which might change depending on \( X_1 \), and so on.
- This is different than Shannon-Fano-Elias coding which uses the same interval length at each step.
- Thus, if a symbol gets very probable, it uses a long relative interval (few bits), and if it gets very improbable, it uses short relative interval (more bits).

How to code? Let \( i \) be the current source symbol number for \( X_i \).

- We maintain a lower and an upper interval position.

  \[
  L_n(i | x_1, x_2, \ldots, x_{n-1}) = \sum_{j=1}^{i-1} p(x_n = j | x_1, x_2, \ldots, x_{n-1}) \quad (8)
  \]

  \[
  U_n(i | x_1, x_2, \ldots, x_{n-1}) = \sum_{j=1}^{i} p(x_n = j | x_1, x_2, \ldots, x_{n-1}) \quad (9)
  \]

- on arrival of \( n^{th} \) input symbol, we divide the \((n-1)^{st}\) interval which is defined by \( L_n \) and \( U_n \) via the half-open interval \([L_n, U_n)\).
Interval Divisions

Example: initial interval is $[0, 1)$ and we divide it depending on the symbol we receive.

$$a \leftrightarrow [L_1(a), U_1(a)] = [0, p(X_1 = a)] \quad (11)$$
$$e \leftrightarrow [L_1(e), U_1(e)] = [p(X_1 = a), p(X_1 = a) + p(X_1 = e)] \quad (12)$$
$$i \leftrightarrow [L_1(i), U_1(i)] = [p(a) + p(e), p(a) + p(e) + p(i)] \quad (13)$$
$$o \leftrightarrow [L_1(o), U_1(o)] = [p(a) + p(e) + p(i), p(a) + p(e) + p(i) + p(o)]$$
$$u \leftrightarrow [L_1(u), U_1(u)] = \left[ \sum_{x \in \{a,e,i,o\}} p(x), \sum_{x \in \{a,e,i,o,u\}} p(x) \right] \quad (14)$$
$$! \leftrightarrow [L_1(!), U_1(!)] = \left[ \sum_{x \in \{a,e,i,o,u\}} p(x), 1 \right] \quad (15)$$

In general, we use an algorithm for the string $x_1, x_2, \ldots$ to derive the intervals $[\ell, u)$ at each time step where $\ell$ is the lower and $u$ is the upper range.

Algorithm

Suppose we want to send $N$ source symbols. Then we can follow the algorithm below.

1 $\ell \leftarrow 0$ ;
2 $u \leftarrow 1$ ;
3 $p \leftarrow u - \ell$ ;
4 for $n = 1 \ldots N$ do
   /* First compute for all $i \in X$, $U_n$ and $L_n$ */
   5 $u \leftarrow \ell + pU_n(x_n|x_1, \ldots, x_{n-1})$ ;
   6 $\ell \leftarrow \ell + pL_n(x_n|x_1, \ldots, x_{n-1})$ ;
   7 $p \leftarrow u - \ell$ ;
**Encoding**

- Once we have final interval, to encode we simply send any binary string that lives in the interval \([\ell, u]\) after running the algorithm.
- On the other hand, we can make the algorithm online, so that it starts writing out bits in the interval once they are known unambiguously.
- Analogous to Shannon-Fano-Elias coding, if the current interval is \([0.100101, 0.100110]\) then we can send the common prefix 1001 since that will not change.

**Example**

Here is an example (Let \(\Box\) be a termination symbol):

<table>
<thead>
<tr>
<th></th>
<th>(p(a) = 0.425)</th>
<th>(p(b) = 0.425)</th>
<th>(p(\Box) = 0.15)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(p(a</td>
<td>b) = 0.28)</td>
<td>(p(b</td>
</tr>
<tr>
<td></td>
<td>(p(a</td>
<td>bb) = 0.21)</td>
<td>(p(b</td>
</tr>
<tr>
<td></td>
<td>(p(a</td>
<td>bbb) = 0.17)</td>
<td>(p(b</td>
</tr>
<tr>
<td></td>
<td>(p(a</td>
<td>bbba) = 0.28)</td>
<td>(p(b</td>
</tr>
</tbody>
</table>

- With these probabilities, we will consider encoding the string \(bbba\Box\), and we’ll get the final interval

\[
\begin{array}{c}
\text{bbba} \Box \\
\text{10111101} \\
\text{1011110} \\
\text{10011111} \\
\text{10100000} \\
\text{100111101}
\end{array}
\]

- I.e., the final code word will be 100111101
- Let’s look at the entire picture
Coding Example from D.J.C. MacKay’s 2001 book.

Q: Why can’t we use 1001111?  
A: Because its interval is too large.  
Codeword 100111101’s interval is entirely within bbba□’s interval, so we are prefix free.

Decoding

To decode a binary string, say \( \alpha = 0.z_1 z_2 z_3 \ldots \) we use algorithm:

1. \( \ell \leftarrow 0 \);
2. \( u \leftarrow 1 \);
3. \( p \leftarrow u - \ell \);
4. while special symbol □ is not received do
5. find \( i \) such that:
   \[
   L_n(i|x_1, \ldots, x_{n-1}) \leq \frac{\alpha - \ell}{u - \ell} < U_n(i|x_1, \ldots, x_{n-1})
   \]
   \[u \leftarrow \ell + pu_n(i|x_1, \ldots, x_{n-1})\]
   \[\ell \leftarrow \ell + pu_n(i|x_1, \ldots, x_{n-1})\]
5. \( p \leftarrow u - \ell \);
Number of bits

- Problem is, a given number in the final interval \([L_n, U_n]\) could be arbitrarily long (e.g., repeated or irrational number). We only need to send enough to uniquely identify string.
- How do we choose the number of bits to send?
- Define
  \[
  F_n(i|x_1, x_2, \ldots, x_{n-1}) = \frac{1}{2}[L_n(i) + U_n(i)]
  \]  
  (17)
  and \([F_n(i|x_1, x_2, \ldots, x_{n-1})]_\ell\) which is \(F_n\) truncated to \(\ell\) bits.
- We could use \(\ell(x_n|x_1, \ldots, x_{n-1}) = \lceil \log 1/p(x_n|x_1, \ldots, x_{n-1}) \rceil + 1\)
- Instead, let's use the Shannon length of the entire code as
  \[
  \ell(x_1:n) = \lceil \log 1/p(x_1:n) \rceil + 1
  \]  
  (18)

Code length

- By the same arguments we made for the Shannon-Fano-Elias codes, this is a prefix code and thus is uniquely decodable, etc.
- Also, we have:
  \[
  E\ell(x_1:n) = \sum_{x_1:n} p(x_1:n)\ell(x_1:n)
  \]
  (19)
  \[
  = \sum_{x_1:n} p(x_1:n) \left( \lceil \log 1/p(x_1:n) \rceil + 1 \right)
  \]  
  (20)
  \[
  \leq - \sum_{x_1:n} p(x_1:n) \log p(x_1:n) + 2
  \]
  (21)
  \[
  = H(x_1:n) + 2
  \]  
  (22)
- So the per symbol length \(\leq H(X_1:n)/2 + 2/n \rightarrow H(X)\)
- But this was not a block code.
### Estimating $p(x_n|x_1, \ldots, x_{n-1})$

- We still have the problem that we need to estimate $p(x_n|x_1, \ldots, x_{n-1})$.
- We’d like to use adaptive models.
- One possibility is the Dirichlet model, having no independencies:
  
  $$p(a|x_1:n-1) = \frac{N(a|x_{1:n-1}) + \alpha}{\sum_{a'}(N(a'|x_{1:n-1}) + \alpha)}$$  

  (23)

- Small $\alpha$ means more responsive.
- Large $\alpha$ means more sluggish.
- How do we derive this? We can do so in a Bayesian setting.
- In general the problem of density estimation is a topic in and of itself.

### Laplace’s rule: Bayesian derivation

- For simplicity, assume binary alphabet, so $\mathcal{X} = \{0, 1\}$.
- $N_0 = N(0|x_{1:n})$ and $N_1 = N(1|x_{1:n})$ counts, and $N = N_0 + N_1$.
- Assumptions: there exists $p_0$, $p_1$ and $p_\square$ for 0, 1, and the termination symbol.
- Length of a given string has a geometric distribution. I.e.,

  $$p(\ell) = (1 - p_\square)^\ell p_\square$$  

  (24)

  so that $E\ell = 1/p_\square$ and $\text{var}(\ell) = (1 - p_\square)/p_\square^2$.
- Characters are drawn independently, so that

  $$p(x_{1:n}|p_0, N) = p_0^{N_0} p_1^{N_1}$$  

  (25)

  this is the “likelihood” function of the data.
- Uniform priors, i.e., $\Pr(p_0) = 1$ for $p_0 \in [0, 1]$. 
Laplace’s rule: Bayesian derivation

- Then

\[
\Pr(p_0|x_{1:n}, N) = \frac{\Pr(x_{1:n}|p_0, N)\Pr(p_0)}{\Pr(x_{1:n}|N)} \tag{26}
\]

\[\Rightarrow \Pr(p_0|x_{1:n}, N) = \frac{p_0^{N_0}p_1^{N_1}}{\Pr(x_{1:n}|N)} \tag{27}\]

- Using the Beta integral, we can get:

\[
\Pr(x_{1:n}|N) = \int_0^1 \Pr(x_{1:n}, p_0|N)dp_0 = \int_0^1 p_0^{N_0}p_1^{N_1}\Pr(p_0)dp_0 \tag{28}
\]

\[= \frac{\Gamma(N_0 + 1)\Gamma(N_1 + 1)}{\Gamma(N_0 + N_1 + 2)} = \frac{N_0!N_1!}{(N_0 + N_1 + 1)!} \tag{29}\]

To make a prediction, we want:

\[
\Pr(X_{n+1} = 0|x_{1:n}, N) = \int_0^1 \Pr(0|p_0)\Pr(p_0|x_{1:n}, N)dp_0 \tag{30}
\]

\[= \int_0^1 p_0^{N_0}p_1^{N_1}\Pr(x_{1:n}|N)dp_0 \tag{31}\]

\[= \int_0^1 p_0^{N_0+1}p_1^{N_1}\Pr(x_{1:n}|N)dp_0 \tag{32}\]

\[= \left[\frac{(N + 0 + 1)!N_1!}{(N_0 + N_1 + 2)!}\right] / \left[\frac{N_0!N_1!}{(N_0 + N_1 + 1)!}\right] \tag{33}\]

\[= \frac{N_0 + 1}{N_0 + N_1 + 2} = \text{Laplace’s rule} \tag{34}\]

- Dirichlet’s approach is \((N_0 + \alpha)/(N_0 + N_1 + 2\alpha)\) so the only difference is the fractional \(\alpha\) (Exercise: derive Dirichlet’s formula).