Information Theory (Compression) Review Sheet prepared by David Warde-Farley (dwf-at-cs.toronto.edu)

Notation $\mathcal{A}_X$ and $\mathcal{A}_Z$ are source and code alphabets.
$\mathcal{A}_X^+$ and $\mathcal{A}_Z^+$ are sequences of one or more symbols.
A symbol code C is a mapping $\mathcal{A}_X \rightarrow \mathcal{A}_Z^+$, use concatenation to extend this to a mapping for the extended code, $C^+ : \mathcal{A}_Z^+ \rightarrow \mathcal{A}_Z^+$

$$c^+(x_1x_2x_3\ldots x_N) = c(x_1)c(x_2)c(x_3)\ldots c(x_N)$$

A code is *uniquely decodable* if $C^+\mathcal{A}_X^+ \rightarrow \mathcal{A}_Z^+$ is one-to-one, i.e. $\forall \mathbf{x}, \mathbf{x}' \in \mathcal{A}_X^+, \mathbf{x} \neq \mathbf{x}' \Rightarrow c^+(\mathbf{x}) \neq c^+(\mathbf{x}')$
A code is *instantaneously decodable* if $\forall \mathbf{x}, \mathbf{x}' \in \mathcal{A}_X^+, \mathbf{x}$ not a prefix of $\mathbf{x}'$, $\mathbf{z} = c(\mathbf{x})$ not a prefix of $\mathbf{z}' = c(\mathbf{x}')$

The Sardinas-Patterson Theorem tells us how to check whether a code is uniquely decodable. Let C be the set of codewords. Define $C_0 = C$. For $n > 0$, define

$$\begin{aligned} C_n &= \{w \in \mathcal{A}_X^+ | uw = v \text{ where } u \in C, v \in C_{n-1}, \\ &\quad \text{or } u \in C_{n-1}, v \in C\} \\ C_\infty &= C_1 \cup C_2 \cup C_3 \ldots \end{aligned}$$

Theorem: the code C is uniquely decodable if and only if C and C_∞ are disjoint. This, of course, is completely useless in practice, unless you've got an infinite amount of time on your hands.

Kraft/McMillan Inequality (McMillan proved for uniquely decodable codes, Kraft for instantaneous)
There is an *instantaneous* binary code with codewords having $l_1, l_2, \ldots l_I$ if and only if

$$\sum_{i=1}^{I} \frac{1}{2^{l_i}} \leq 1$$

A good code minimizes the expected codeword length L where

$$L = L(C, X) = \sum_{i=1}^{I} p_i l_i$$

Entropy The entropy of a source (absolute lower bound on the average per-symbol codeword length of *any* uniquely decodable code) with marginal symbol distribution p is

$$H(X) = \sum_i p_i \log_2 \frac{1}{p_i}$$

The entropy of the kth extension of a source X, denoted X^k (all strings from $\mathcal{A}_X^+$ of length k) is

$$H(X^k) = kH(X)$$

Shannon-Fano Codes Construct codes such that codewords for symbols $a_1, a_2, \ldots a_I$ with probabilities $p_1, p_2, \ldots p_I$ have code lengths

$$l_i = \lceil \log_2(1/p_i) \rceil$$

Kraft-McMillan says such a code exists since

$$\sum_{i=1}^{I} \frac{1}{2^{l_i}} \leq \sum_{i=1}^{I} \frac{1}{2^{\log_2(1/p_i)}} = \sum_{i=1}^{I} p_i = 1$$

Expected length of a Shannon-Fano code is $1+H(X)$ (because $\lceil \log_2(1/p_i) \rceil < 1 + \log_2(1/p_i)$)

Huffman Codes are actually optimal single-symbol codes, but we can't actually do better than within 1 bit of the entropy unless we code symbols in blocks.

Shannon's Noiseless Coding Theorem For any desired average length per symbol R, there is a value of N for which a uniquely decodable binary code for X^N exists that has expected length less than NR.
Intuitively this is because any optimal or nearly optimal (Shannon-Fano) code for blocks of N symbols will have expected codeword length $L_N \leq 1 + H(X^N)$, making the expected length per original source symbol $\frac{L_N}{N} = H(X) + \frac{1}{N}$.

Typical Sets Another way to construct optimal codes is to use codes of a fixed length which cannot always be correctly decoded.
Assume we are coding blocks of length N. The Law of Large Numbers tells us that the sequence $a_{i_1}, \ldots a_{i_N}$ is very likely to be a "typical" one where

$$\frac{1}{N} \log_2(1/(p_{i_1}, p_{i_2}, \ldots p_{i_N})) \leq H(X) + \eta/\sqrt{N}$$

and the probability of such a sequence will satisfy

$$p_{i_1} \ldots p_{i_N} \geq 2^{-NH(X)-\eta\sqrt{N}}$$

Scale as $1/\sqrt{N}$ because that's how a standard deviation of an average scales. Chebychev's inequality tells us that most sequences will satisfy this condition for some large enough η. The total probability for all such sequences can't be greater than one thus the number of such sequences can't be greater than $2^{NH(X)+\eta\sqrt{N}}$. We can encode these in NR bits if $NR > NH(X) + \eta\sqrt{N}$ (using any arbitrary mapping from sequences to one of the 2^{NR} codes). Define $H_\delta(X^N)$ as the average codeword length needed for the typical set to leave out only δ of all sequences. **For large N, H_δ becomes almost independent of δ.**

Another Statement of Shannon's Theorem
Let X be an ensemble with entropy $H(X) = H$ bits. Given any $\epsilon > 0$ and $0 < \delta < 1$ there is some positive integer N_0 such that for $N > N_0$

$$H - \epsilon < \frac{1}{N} H_\delta(X^N) < H + \epsilon$$

Even if we want extremely low compression error probability (δ), the average codeword length $\frac{1}{N}H_\delta(X^N)$ doesn't have to exceed $H + \epsilon$ bits. On the other hand, even if our error probability is close to one, we still need at least $H - \epsilon$ bits. Tiny tolerance for error gets us from $H + 1$ to $H + \epsilon$ bits. If we need error-free decoding just specify one bitstring as the prefix for the unencodables and use any UD code after that. They happen so infrequently so it won't affect things at all.

Arithmetic Coding works by subdividing the interval on $[0, 1)$, subdividing the ith interval when a_i is encoded, and so forth. We encode by transmitting a bit string that defines a point on the interval. We can pick a codeword of length k for a block with $p = v - u$ (difference of top of interval and bottom of interval) if $k \geq \log_2(1/p) + 1$, so no longer than $\lceil \log_2(1/p_b) \rceil + 1 < \log_2(1/p_b) + 2$.

Average codeword length of encoding the Nth extension in this fashion is

$$2 + \sum_b p_b \log_2(1/p_b) = 2 + H(\mathcal{A}_X^N) = 2NH(\mathcal{A}_X)$$

Per symbol of $\mathcal{A}_X$ we're transmitting, on average, less than $H(\mathcal{A}_X) + 2/N$. This is superior to using a Huffman code specifically because we don't have to store probability tables for every possible block of length N.

We use **Stream Coding** to transmit the bits as soon as they're determined, eliminating the need for block lengths and the whole idea of blocks. Once we transmit a bit we "expand the interval" (if we're in the bottom half, double both boundaries, otherwise subtract a half and then double), move the radix point one place right. This eliminates the need for arbitrarily high precision for storing the interval boundaries. To get around the problem of intervals that include 1/2, we can expand the interval *around the middle* of the range ($u - 1/4$ and $v - 1/4$) and *remember* to transmit an "opposite" bit, or several. Using fixed point is probably better on most machines. Using Huffman can be way faster and almost as good.

KL Divergence Penalty for guessing the probability model wrong (assuming we're using a near-entropy encoding method): relative entropy/KL divergence (q = model distribution, p = real distribution)

$$\sum_{i=1}^{I} p_i \log_2(1/q_i) - \sum_{i=1}^{I} p_i \log_2 1/p_i = \sum_{i=1}^{I} p_i \log_2(p_i/q_i)$$

Asymmetric, non-negative. Estimating probabilities from data and sending them can't be optimal because then the code isn't complete (all sequences of code bits are possible).

Adaptive Models continually re-estimate the probabilities using counts of symbols in the earlier part. We add 1 to all the counts to avoid giving symbol 0 probability and a codeword length of ∞. Dynamic re-estimation easy to do with arithmetic codes (subdivide according to current probabilities) but much harder to do with [blocked] Huffman codes.

Any adaptive model also assigns probability to every sequence of symbols, multiplying the probabilities as we go. **With an optimal method, the number of bits used to encode the entire sequence will be close to the log(1/p(sequence)).** Generally,

$$P(X_n = a_i) = \frac{1 + \text{Number of times we've seen } a_i}{I + n - 1}$$

I is size of source alphabet. Called "Laplace's rule of succession". Probability of a sequence of n symbols is (where n_i is number of times a_i occurs in sequence)

$$\frac{(I-1)!}{(I+n-1)!} \prod_{i=1}^{I} n_i!$$

Kth order Markov sources depend on the previous K symbols. Probability of $x_1, x_2, \ldots x_n$ with $K = 2$:

$$\begin{aligned} P(x_1 x_2 \ldots x_n) &= P(x_1 = a_{i_1}) \times \\ &\quad P(x_2 = a_{i_2} | x_1 = a_{i_1}) \times \ldots \\ &\quad P(x_n = a_{i_n}) P(x_{n-1} = a_{i_{n-1}}, x_{n-2} = a_{i_{n-2}}) \\ &= P(x_1) P(x_2|x_1) M(i_2, i_3, i_4) \ldots \\ &\quad M(i_{n-2}, i_{n-1}, i_n) \end{aligned}$$

$M(i, j, k)$ is the probability of symbol a_k after a_i and a_j. Estimating the Markov transition probabilities, accumulating frequencies $F(i, j, k)$,

$$M(i, j, k) = F(i, j, k) / \sum_{k'} F(i, j, k')$$

Prediction by Partial Match is like a variable-order Markov model. Maintains frequencies for characters that have been seen before in *all* contexts that have occurred before up to some maximum order. If we need to encode a character that we've never seen before in the context we're using we transmit an "escape" flag and go up one order. If we've never seen it before we end up in "order - 1" where everything has $F = 1$. Can either count a character in a context every time or only when it does not appear in a higher-order context.

Efficient to use a **trie** (an ordered prefix tree, position of node denotes the key of the dictionary, lookup for key of length ℓ is $O(\ell)$ time). PPM implicitly learns a vocabulary so it makes for very good learning of English text, etc., but more practical method is to store dictionary explicitly. Big disadvantage of probabilistic models is computational complexity.

LZ77uses the past text as the dictionary, buffer of size W that contains the previous S characters and the following $W - S$ characters. Encode up to $W - S$ characters by sending a pointer p to a past character and number of characters n to take from the buffer. Choose p to maximize n (and if possible minimize p so we can encode it with less bits). Transmit `NULL` for p on dictionary miss and don't transmit n, and just transmit the symbol raw. Could output p as sequences of one or more bytes or use Huffman or arithmetic coding for pointers. Speed up buffer lookup with hash table. `gzip` builds hash table for all strings of length three then searches the hash bucket for the next three characters to find the longest match.

`gzip` uses a variant called "deflate", finds duplicated strings in blocks of the input data. The second occurrence is replaced by pointer, length pair. Distances limited to 32K bytes, lengths limited to 258 bytes. When a string does not occur in previous 32K it is emitted literally. Literals and match lengths compressed with one Huffman tree, pointers with another. The trees are stored in a compact form at start of block. Duplicated strings found using hash table. **LZ78** came after where dictionary is kept explicitly, entire past text used. **LZW** variant used in `compress` and `gif`: start with dictionary containing just alphabet, encode form current position by

- Find longest possible match of characters following current position with dictionary item Transmit index of that dict. item
- Add the matched phrase `plus the character` to the dictionary
- Set current position to the character following match

Dictionary codes get longer but slowly. LZW proven to asymptotically compress down to entropy of any ergodic source (memoryless on sufficient timescale). This is why it is called a "universal" compression algorithm.