CPS290: Algorithmic Foundations of Data Science **January 18, 2017**

Lecture 1 : Data Compression and Entropy

Lecturer: Kamesh Munagala *Scribe: Kamesh Munagala*

In this lecture, we will study a simple model for data compression. The compression algorithms will be constrained to be "lossless" meaning that there should be a corresponding decoding algorithm that recovers the original data exactly. We will study the limits of such compression, which ties to the notion of entropy. We will also study a simple algorithm for compression when the input text arrives one symbol at a time.

Prefix Codes

A compression model is specified as a probabilistic model for generating input text. In the most basic model, there is a set $\{a_1, a_2, \ldots, a_m\}$ of m symbols. We assume there is an underlying distribution D over these symbols, where symbol a_i has associated probability p_i. Note that $\sum_{i=1}^{m} p_i = 1$, since D is a distribution. The input text of length n is generated by choosing each symbol independently from D. We typically assume n is large compared to m.

The standard way of representing such text would be to encode each a_i in binary using $q_0 = \lceil \log_2 m \rceil$ bits. The quantity q_0 is the number of bits per symbol in this encoding. For instance, ASCII is such an encoding of text. The decoding algorithm knows the mapping of symbols to binary strings, and knows the length q_0 of each encoded symbol. This is sufficient to decode a binary string corresponding to the concatenation of the encodings of each character of the input text.

Such an encoding can be sub-optimal in terms of the number of bits per symbol required. Let us make this statement more precise. Any code is a mapping of each of the m input symbols to a binary string. Suppose ℓ_i is the length of the binary string corresponding to symbol a_i. Then, on average, the length of the encoding of an input symbol is $\sum_{i=1}^{m} p_i \ell_i$. This is the *expected* length of the encoded string if a symbol is drawn at random from the distribution D.

Note that we cannot allow symbols to be mapped to arbitrary binary strings. Such a mapping cannot be uniquely decoded. For instance, suppose a_1 maps to 10, a_2 to 1 and a_3 to 101, then if the output is 101, it is not clear if it is an encoding of the text $a_1 a_2$ or the text a_3. This leads to the notion of a *prefix code*. A prefix code has the property that there is no pair of symbols such that the encoding of one symbol is a prefix of the encoding of the other symbol. We will now show that a code is uniquely decodable iff it is a prefix code.

To see this, we can view any code as a binary tree, where the left branch at any node is labeled 0 and the right branch 1. The leaves of this tree are annotated with the symbols. To encode a symbol, we read off the binary string formed by traversing the path from the root to the corresponding leaf. It is clear that such a code is a prefix code. Decoding such a code is unique, since the decoder, when presented with the same tree, and a long binary sequence, traverses the tree starting at the root based on the binary sequence. When it encounters a leaf, it outputs the corresponding symbol, and goes back to the root. Conversely, if a code is not a prefix code, then there is an internal node of the tree that corresponds to a symbol. In this case, when the prefix is encountered, the decoder does not know whether to output the symbol or traverse further to a leaf.

Huffman Codes

The goal now is to construct a prefix code with smallest expected length per symbol. This corresponds to finding a binary tree on m leaves whose expected root to leaf path length is smallest. Without loss of generality, assume $p_1 \leq p_2 \leq p_3 \leq \cdots \leq p_m$. Ideally, the code should place a_1 deepest in the tree and a_m the shallowest. We will show a greedy algorithm that roughly does this.

Claim 1. *Let $f(p_1, p_2, \ldots, p_m)$ denote the expected length per symbol of the optimal prefix code. Suppose $p_1 \leq p_2 \leq \cdots \leq p_m$. Then $f(p_1, p_2, \ldots, p_m) = p_1 + p_2 + f(p_1 + p_2, p_3, \ldots, p_m)$.*

Proof. Consider any binary tree, and a node v in the tree. Let S denote the leaves in the subtree corresponding to v. Let $w_v = \sum_{i \in S} p_i$. Then it is easy to check that the expected length of the tree is equal to the sum of the w_v over all nodes v in the tree.

This means that if i and j are two leaves with a common parent in the optimal tree, then we can remove i and j and pretend its parent is a leaf with frequency $p_i + p_j$ – the resulting smaller tree is a prefix code for the smaller set of symbols, and should be the optimal such tree. Therefore, if $f(p_i, p_j, p_1, \ldots, p_m) = p_i + p_j + f(p_i + p_j, p_1, \ldots, p_m)$, if i and j are siblings in the optimal tree.

We will finally show that a_1 and a_2, the symbols with the smallest frequencies, are siblings. This follows from an exchange argument. Suppose they are not. Consider the two deepest nodes in the optimal tree. Swap a_1 and a_2 with these nodes. This only reduces the expected length of the tree. □

The above claim suggests a greedy algorithm to find the optimal prefix code – find the symbols with the smallest frequencies, make them siblings, and replace them with a symbol whose frequency is equal to the sum of their frequencies. Now recurse on the smaller instance. This is the Huffman encoding algorithm, and can be implemented in $O(m \log m)$ time using a priority queue to store the frequencies of the current set of symbols.

Discrete Entropy

The above construction is algorithmic – it shows the existence of an optimal prefix code. The expected code length captures how "compressible" the distribution over symbols is. If the distribution is very skewed, then so is the Huffman tree, and the expected code length is small. Conversely, if all symbols have the same frequency, then the simple ASCII encoding is optimal.

The expected code length has a closed form to a very good approximation. This closed form is a fundamental property of distributions, and is termed *entropy*. It captures how much randomness there is in the distribution – the more the entropy, the greater the length of the encoding, and the less compressible is text drawn from this distribution.

Why do we need a closed form? It helps us get a handle on understanding how good various practical compression schemes are. We will present an example later.

We begin with a theorem that essentially says that in a prefix code, all the codewords cannot be really small in length. If some codewords have very small lengths, there must be others with really large lengths. This is very intuitive – we cannot construct binary trees where all leaves are shallow.

Theorem 1 (Kraft's Inequality)**.** *Suppose there is a prefix code where the codewords for $a_1, a_2, \ldots, a_m$ have lengths $\ell_1, \ell_2, \ldots, \ell_m$. Then*

$$\sum_{i=1}^{m} 2^{-\ell_i} \leq 1$$

Conversely, if a collection $\{\ell_i\}$ of integers satisfies the above inequality, then there is a prefix code with those lengths.

Proof. Let $\ell_{\max} = \max_{i=1}^{m} \ell_i$. Consider a complete binary tree of length $\ell_{\max}$. A prefix code corresponds to a subtree of this tree with the same root. Consider a leaf at depth ℓ_i in the Huffman tree. When mapped to the complete binary tree, this leaf is an internal node at depth ℓ_i, and its subtree has depth $\ell_{\max} - \ell_i$, and hence $2^{\ell_{\max}-\ell_i}$ leaves. Different leaves of the Huffman tree map to disjoint subtrees in the complete binary tree. By simple counting, the total number of leaves in these subtrees cannot be more than the total number of leaves in the complete binary tree. Therefore,

$$\sum_{i=1}^{m} 2^{\ell_{\max}-\ell_i} \leq 2^{\ell_{\max}}$$

This implies Kraft's inequality.

To see the converse, suppose Kraft's inequality holds. Consider the complete binary tree of depth $\ell_{\max}$. We will construct a Huffman tree as follows: Suppose $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_m$. Place symbol a_1 and depth ℓ_1 and delete the entire subtree beneath this node. Now place symbol a_2 at any remaining node at depth ℓ_2, delete its subtree, and so on. We will show this process does not fail. Suppose on the contrary that when we try to place symbol a_j, there is no remaining node at depth ℓ_j. This implies we have deleted more than 2^{ℓ_j} nodes of the complete binary tree at this depth. But a_1 deletes $2^{\ell_j-\ell_1}$ nodes, and so on. This means:

$$\sum_{i=1}^{j} 2^{\ell_j-\ell_i} > 2^{\ell_j}$$

This contradicts Kraft's inequality. Therefore, the process terminates with a valid prefix code that assigns symbol a_j a binary string of length exactly ℓ_j. □

We now present a lower bound on the expected code length per symbol via the concept of *discrete entropy.*

Definition 1. *Given a discrete distribution $X = (p_1, p_2, \ldots, p_m)$ on m items, the entropy of this distribution $H(X)$ is defined as:*

$$H(X) = \sum_{i=1}^{m} p_i \log_2 \frac{1}{p_i}$$

Consider the case where $m = 2$. In this case, suppose $\Pr[X = a_1] = p$ and $\Pr[X = a_2] = 1 - p$, then $H(X) = p \log_2 \frac{1}{p} + (1-p) \log_2 \frac{1}{1-p}$. It is easy to check that this is a concave function, which is 0 when $p = 0$ or $p = 1$, and is maximized when $p = 1/2$, where it takes value $\log_2 2 = 1$. In general, the discrete entropy is largest when $p_1 = p_2 = \cdots = p_m = \frac{1}{m}$, where its value is $\log_2 m$. We will now show that discrete entropy is an exact estimate of the expected length of the optimal prefix code, modulo rounding error.

Theorem 2. *Given a distribution $X = (p_1, p_2, \ldots, p_m)$ on m symbols, the Huffman code has length at least $H(X)$ and at most $H(X) + 1$.*

Proof. To show the lower bound, we will minimize the expected code length subject to Kraft's inequality. Suppose the optimal code assigns length ℓ_i to symbol a_i. Then, the expected length of the encoding is $\sum_{i=1}^{m} p_i \ell_i$. But since this is a prefix code, the $\{\ell_i\}$ satisfy Kraft's inequality.

This means the following optimization problem is a lower bound on the length of the Huffman code, where the variables are $\{\ell_i\}$.

$$\text{Minimize} \quad \sum_{i=1}^{m} p_i \ell_i$$

$$\begin{aligned} \textstyle\sum_{i=1}^{m} 2^{-\ell_i} &\leq 1 \\ \ell_i &\geq 0 \quad \forall i = 1, 2, \ldots, m \end{aligned}$$

Note that the optimal code also has the constraint that the ℓ_i are integers. If we ignore that constraint, we only obtain a lower value to the optimization problem. We will show later that the estimate we get is not that bad.

Without further ado, let's solve this optimization problem. It has one objective function and one non-trivial constraint. Let's change the variables to make the constraint simple. Set $x_i = 2^{-\ell_i}$ so that $\ell_i = \log_2 \frac{1}{x_i}$. Then the optimization problem can be rewritten as:

$$\text{Minimize} \quad \sum_{i=1}^{m} p_i \log_2 \frac{1}{x_i}$$

$$\begin{aligned} \textstyle\sum_{i=1}^{m} x_i &\leq 1 \\ x_i &\geq 0 \quad \forall i = 1, 2, \ldots, m \end{aligned}$$

Let $f_i(x_i) = p_i \log_2 \frac{1}{x_i}$. Then the objective is of the form $\sum_{i=1}^{m} f_i(x_i)$. Note that $f_i(x_i)$ is a decreasing function of x_i. Furthermore, the gradient $f_i'(x_i) = -\frac{p_i}{x_i \log_e 2}$ is an increasing negative function in the range $x_i \in [0, 1]$.

The optimum solution $\{x_i^*\}$ can be found by the following argument: Suppose $f_i'(x_i^*) < f_j'(x_j^*)$. In this case, suppose we increase x_i^* by an infinitesimal quantity δ, and decrease x_j^* by the same amount, then the objective decreases by $\delta(f_j'(x_j^*) - f_i'(x_i^*)) > 0$, contradicting the optimality of x^*. Therefore, at the optimum solution, the gradients $f_i'(x_i^*)$ are equal for all i. Since $f_i'(x_i) = -\frac{p_i}{x_i \log_e 2}$, this means $x_i^* \propto p_i$. But $\sum_{i=1}^{m} x_i^* = 1$ by Kraft's inequality, which implies $x_i^* = p_i$. This implies $\ell_i^* = \log_2 \frac{1}{p_i}$, so that the optimal value of the objective $\sum_i p_i \ell_i$ is the entropy $H(X)$.

To show an upper bound, note that $\ell_i = \lceil \log_2 \frac{1}{p_i} \rceil$ satisfies Kraft's inequality. Since $\lceil \log_2 \frac{1}{p_i} \rceil \leq \log_2 \frac{1}{p_i} + 1$, this means $\sum_i p_i \ell_i \leq H(X) + \sum_i p_i = H(X) + 1$. Therefore, the optimal Huffman tree has length at most $H(X) + 1$, completing the proof. □

A Simple Compression Scheme and its Analysis

In reality, we need to compress large volumes of text in one pass over the text. In such settings, it is infeasible to first estimate the frequencies of different symbols and then construct an optimal Huffman tree. There are several compression algorithms that adjust the encoding of symbols as time goes by, based on their current frequencies. Such algorithms have the additional advantage that they can exploit patterns in the data beyond simple frequencies of symbols. We present one such scheme – this scheme is not only close to optimal in terms of symbol entropy in the worst case (as we will prove below), but is also quite practical and simple to analyze.

Move-to-front (MTF) coding maintains the symbols in a linked list L. At time t, suppose the encoder sees symbol a. It walks down L until it encounters a. Suppose it is the i^{th} symbol in the list. Then the encoder emits $\lceil \log_2 i \rceil$ 0's followed by the binary representation of the number i (which takes $\lceil \log_2 i \rceil$ bits). The encoder moves a to the front of the linked list.

The decoder maintains the same ordered list as the encoder. The decoder walks down the compressed binary string till it encounters a 1. Suppose it saw q zeros. It interprets the next q bits as the binary representation of the position of the next symbol in the list L. It walks down the list to that position, emits that symbol, and moves it to the front of its list. It then continues parsing the compressed binary string.

It is an easy exercise to show that the behavior of the decoder is correct. One may think that emitting $\lceil \log_2 i \rceil$ 0's is wasteful. But note that there is a need to separate the encodings of different symbols. A prefix code is one way of doing this if the decoder has access to the same code. A different way is to have the same bit length for all codewords, but this does not achieve any compression. MTF will assign small codewords to highly repetitive symbols, but at the expense of doubling their length. We show later that we can in fact do better than double the code length using roughly the same ideas.

Worst-case Analysis of MTF

Suppose the text contains N symbols, where symbol a_i appears with frequency p_i. Let $X = (p_1, p_2, \ldots, p_m)$. An optimal *static* Huffman code is one that uses these frequencies to construct a fixed code. Such a code uses roughly $H(X)$ bits per symbol. We will show that MTF is not much worse – it uses at most $2(H(X) + 1)$ bits per symbol. Note that since MTF is a *dynamic code*, meaning that it changes the encoding of any symbol as time goes by, it can in fact do much better than a static Huffman code. The analysis below shows that in no case can be more than a factor of 2 worse than the Huffman code. (The factor 2 is not necessary, as we show later.)

Before presenting the proof, to gain intuition, suppose we want to make MTF as bad as possible by carefully constructing text. Suppose at each step, we try to make MTF encode using the largest possible integer, m. This means the next symbol in the text must always be the current last symbol in the linked list. The only way of always doing this is a pattern of the form $a_1 a_2 \ldots a_m a_1 a_2 \ldots a_m a_1$ and so on, assuming the initial list had a_1 at the very end, a_2 second last, and so on. In this case, MTF always outputs the binary encoding of m, so it takes $2N \log_2 m$ bits for N symbols. But for this text, the frequencies of all symbols are all equal to $1/m$, so the entropy is $\log_2 m$. This means Huffman coding also takes at least $N \log_2 m$ bits.

We now present an argument that works for any text. Consider some symbol a_i with frequency p_i. There are Np_i occurrences of this symbol. Suppose the first of these appears at time t_1, the second at time t_2, and so on. Let $m_1 = t_1$, $m_2 = t_2 - t_1$, and so on. Note that $\sum_j m_j = N$, the length of the text. Note further that j takes Np_i values.

Now, the first time a_i is encountered, it is at position m_1, the second time at position m_2, and so on. This means the length of the encoded text corresponding to these occurrences is $2\lceil \log_2 m_1 \rceil, 2\lceil \log_2 m_1 \rceil, \ldots$. Therefore, the total length of the binary encoding for all occurrences of a_i is

$$\sum_j 2\lceil \log_2 m_j \rceil \leq 2 \sum_j \log_2 m_j + 2p_i N = 2Np_i \left(\frac{\sum_j \log_2 m_j}{Np_i} + 1 \right)$$

Since log is a concave function, we can use Jensen's inequality:

$$\frac{\sum_j \log_2 m_j}{Np_i} \leq \log_2 \left(\frac{\sum_j m_j}{Np_i} \right) = \log_2 \frac{1}{p_i}$$

Therefore, the total length of the binary encoding for all occurrences of a_i is at most

$$2Np_i \left(\log_2 \frac{1}{p_i} + 1 \right)$$

Summing this over all symbols a_i, and dividing by N, the average length of a codeword is

$$\sum_{i=1}^{m} 2\left(p_i \log_2 \frac{1}{p_i} + p_i\right) = 2(H(X)+1)$$

Therefore, MTF compression produces compressed text that is at most twice as large as that produced by Huffman coding. Note further that our analysis is *worst-case* – given *any* text with N symbols with frequencies of symbols given by X, the length of the compressed text is close to $2NH(X)$. This does not require that the text satisfies some distributional property like being drawn from independently from distribution X, and so on. In fact, the above bound can be somewhat pessimistic, since MTF can exploit periodic patterns in the data, that is "locality". The entropy bound assumes data is generated independently from an underlying distribution, and there are no patterns beyond the frequencies of the symbols. In reality, patterns abound in data, and different compression schemes exploit different types of patterns.

Elias Codes

The factor of 2 in the above bound is bothersome. It arises because we want to encode integers using a prefix code. The way we did it was to pre-append the binary encoding for i with $q = \lceil \log_2 i \rceil$ zeros. This doubles the length of the encoding. Can we construct a better prefix code? The answer is "yes". We can recurse on the unary encoding of the number of bits in i. Instead of writing out q zeroes, we can write the binary encoding of q, and pre-append it with $r = \lceil \log_2 q \rceil$ zeros. The decoder then reads the string till it sees a 1. Suppose these are r bits It reads the next r bits as the binary representation of q. It then reads the next q bits as the binary representation of i. The number of bits now needed is roughly $\log_2 i + 2\log_2 \log_2 i$, which for large i, beats $2\log_2 i$. With more care, we can replace the ceilings by floors in the above encodings, which saves some more bits. There is no reason to stop here – we can repeat the process, but the gains get progressively less meaningful.