

Course notes for Data Compression - 1
The Statistical Coding Method

Fall 2005

Peter Bro Miltersen

August 29, 2005

Version 2.0

1 The paradox of data compression

Definition 1 Let Σ be an alphabet and let $S \subseteq \Sigma^*$ be a set of possible messages. A lossless *codec* or just code (c, d) consists of a *coder* $C : S \rightarrow \{0, 1\}^*$ and a *decoder* $d : \{0, 1\}^* \rightarrow \Sigma^*$ so that $\forall x \in S : d(c(x)) = x$. The set $c(S)$ is the set of *code words* for the code.

Remark In order to satisfy $d(c(x)) = x$ for all x , c must clearly be injective. Also, if an injective c is given, we can define d appropriately to achieve $d(c(x)) = x$ for all x , so unless we worry about efficiency, we only have to specify a coder in order to have a codec.

Proposition 2 (Compression is impossible) *Let $S = \{0, 1\}^n$. Then, for any codec (c, d) , there exists $x \in S : |c(x)| \geq n$.*

Proof This follows from the pigeon hole principle, indeed, $|S| = 2^n$ and $|\{0, 1\}^{<n}| = 2^n - 1$.

On the other hand,

Proposition 3 (Encyclopedia Britannica can be compressed to 1 bit) *For any message x , there is a codec (c, d) so that $|c(x)| = 1$.*

Proof Assume without loss of generality that $S = \{0, 1\}^*$. Then define c by $c(x) = 0$ and $c(y) = 1y$ for $x \neq y$.

The two propositions suggest that the right approach to studying compression is to try to make codecs compressing many (not just one and not all) strings well. The problem is how to evaluate how well we are doing. It would be nice if we could say that we are doing “optimally” well in some situations - despite Proposition 3 which may suggest otherwise.

There are two conceptual approaches that will allow us to do this.

1. The approach of statistical coding theory. Proposition 3 shows that if the data to be compressed is known in advance, compression is trivial. Thus, the interesting setting is when there is uncertainty about the data to be compressed and probability theory gives us a language for modelling such uncertain situations precisely. In statistical coding theory, we need a *data model*, i.e., a probability distribution over the set of messages S . We want our codec to do well on a random sample from the distribution and measure its performance by the expected length of an encoding of such a random sample.

2. The “restricted code” approach. To rule out codes such as the one from Proposition 3, we restrict our attention to special classes of coders c (memoryless codes, finite memory codes, finite-state codes). Having done this, we may argue that a particular code is optimal among all codes in the class for encoding a fixed string such as, say, Encyclopedia Britannica.

Though the two approaches are conceptually quite different, they often end up proving essentially the same results about the same codes, but from quite different point of views - which may lead to some confusion. The approach of Kieffer’s *Lectures on Source coding* is mainly the restricted code approach. Here, we will talk about the same basic issues as Kieffer from the viewpoint of statistical coding theory though we relate our description to the restricted code point of view in Section 8.

2 Prefix codes

The basic philosophy of statistical coding theory is this: Define a reasonable probability distribution over S (a *data model*) and find a coder c that minimizes $E[|c(x)|]$, i.e., *the expected code length*, for randomly chosen x according to the distribution.

Proposition 2 shows that compression of all messages in $S = \{0, 1\}^n$ is impossible. But what about the expected code length for the basic case of the uniform distribution over S ?

The following example suggests that non-trivial compression is then possible. Let $S = \{0, 1\}^2$, $c(00) = \lambda$, $c(01) = 0$, $c(10) = 1$, $c(11) = 01$. With x chosen uniformly in S , we have that $E(|c(x)|) = 1$ and have seemingly compressed two bits to one on the average!

While this is, in some sense, correct it is not interesting as it does not scale: We cannot, say, compress 4 bits to 2 bits by concatenating the code we have defined with itself (i.e., letting $c(xy) = c(x) \cdot c(y)$ for $x, y \in \{0, 1\}^2$), as the resulting code would not be injective (we would have $c(0011) = c(0110)$). Thus, we seem to be cheating by not demanding that our code words explicitly or implicitly contain an “end symbol” that could allow us to see where they stop. The following definition repairs this flaw of our original definition.

Definition 4 A *prefix code* c has the property that $\forall x, y \in S : x \neq y \Rightarrow c(x)$ is not a prefix of $c(y)$.

Unlike the misbehaving code in the example, a prefix code for an alphabet can be used to define a code for texts, which is a very useful property¹:

Proposition 5 *If c is a prefix code for Σ then c^n is a prefix code for Σ^n where $c^n(x_1, x_2, \dots, x_n) = c(x_1) \cdot c(x_2) \cdot \dots \cdot c(x_n)$ with \cdot denoting concatenation.*

Some alternative ways of viewing a prefix code which shall be very useful for us are the following.

- A prefix code is an assignment of the messages of S to the leaves of a rooted binary tree with the down-going edges from each internal node labelled 0 and 1. We can find the codeword corresponding to a message x by reading the labels on the edges on the path from the root of the tree to the leaf corresponding to x .
- A more subtle interpretation is this. Given a string $a \in 0, 1^*$, we can define the corresponding *dyadic interval* $I_a \subseteq [0, 1)$ by $I_a = [0.a, 0.a1111\dots)$ where $0.a$ and $0.a1111\dots$ are to be interpreted as fractional binary numbers. For instance $I_{101} = [0.101, 0.101111\dots) = [0.101, 0.110)$, or in decimal notation $[5/8, 3/4)$. Then, a prefix code is exactly an assignment of the set of messages to a set of *mutually disjoint* dyadic intervals in $[0; 1)$.

The second way of viewing a code is very useful for proving the following theorem.

Theorem 6 (Kraft-McMillan inequality)

1. *Let m_1, m_2, \dots be the lengths of the code words of a prefix code. Then $\sum 2^{-m_i} \leq 1$.*
2. *Conversely, if m_1, m_2, \dots , are integers satisfying $\sum 2^{-m_i} \leq 1$ then there is a prefix code c so that $\{m_i\}$ are the lengths of the code words.*

Proof

1. If m_1, m_2, \dots , are the lengths of the code words of a prefix code, the lengths of the corresponding dyadic intervals are $2^{-m_1}, 2^{-m_2}, \dots$. As they are disjoint and all fit into $[0, 1)$, we must have $\sum 2^{-m_i} \leq 1$.

¹Note that although c^n is a prefix code for Σ^n for fixed n , we don't get a prefix code on Σ^* by defining $c^*(x_1, x_2, \dots, x_{|x|}) = c(x_1) \cdot c(x_2) \cdot \dots \cdot c(x_{|x|})$, simply because $c^*(x)$ is a prefix of $c^*(y)$ whenever x is a prefix of y .

2. We want to arrange intervals of lengths $2^{-m_1}, 2^{-m_2}, \dots$ disjoint in $[0, 1)$ so they all become dyadic intervals and hence define a prefix code. We do it by first sorting the intervals in decreasing order by length, or, correspondingly, sorting the code word lengths in *increasing* order by length, i.e., we assume WLOG that $m_1 < m_2 < \dots$. Then we simply put the intervals side by side starting with the largest as the leftmost. Now, the interval of length 2^{-m_i} will be assigned the slot $[\sum_{j<i} 2^{-m_j}, \sum_{j<i} 2^{-m_j} + 2^{-m_i})$ which is a dyadic interval as $\sum_{j<i} 2^{-m_j}$ is an integer multiple of 2^{-m_i} .

3 Entropy

Given a probability distribution, the *entropy* of the distribution is, intuitively, a measure of the amount of randomness in the distribution, or, more precisely, the expected degree of your surprise when you receive a sample from the distribution. We shall see that it also measures precisely how well samples from the distribution may be compressed.

Definition 7 Given a fixed probability distribution p on S , i.e., a map $p : S \rightarrow [0, 1]$ so that $\sum_{x \in S} p(x) = 1$, we define the *self-entropy* $H(x)$ of a message x as $-\log_2 p(x)$. The entropy $H[p]$ of the probability distribution is defined as $E[H]$, i.e., $H[p] = -\sum_{x \in S} p(x) \log_2 p(x)$.

Given a stochastic variable or vector X we define its entropy $H[X]$ as the entropy of the underlying distribution, i.e.,

$$H[X] = -\sum_i \Pr[X = i] \log_2 \Pr[X = i].$$

Even though the definition is dimensionless, we often measure entropy in unit “bit” and say that the entropy of a distribution is, say, “3 bits” rather than just “3”.

Some easily proved facts about entropy:

- The entropy of the uniform distribution on $\{0, 1\}^n$ is n bits.
- If X_1 and X_2 are independent stochastic variables then $H(X_1, X_2) = H(X_1) + H(X_2)$.
- For any function f , $H(f(X)) \leq H(X)$ (i.e., entropy cannot be increased deterministically)

The main theorem linking prefix codes with the entropy concept is the following theorem of Shannon (a somewhat simplified version of Shannon's classical lossless source coding theorem).

Theorem 8 *Let S be a set of messages and let X be a random variable with S as support (i.e., the outcomes of X with positive probability is exactly S). We then have:*

1. For all prefix codes c on S , $E[|c(X)|] \geq H[X]$.
2. There is a prefix code c on S so that $E[|c(X)|] < H[X] + 1$.

Proof Let $p(x) = \Pr[X = x]$.

1. $E[|c(X)|] \geq H[X]$ by definition means $\sum_{x \in S} p(x)|c(x)| \geq \sum_{x \in S} p(x) \log_2(1/p(x))$ which is true if and only if $\sum_{x \in S} p(x)[|c(x)| - \log_2(1/p(x))] \geq 0$, i.e., if and only if $\sum_{x \in S} p(x) \log_2(2^{-|c(x)|}/p(x)) \leq 0$, so this we must show.

But by Jensen's inequality, $\sum p(x) \log_2(2^{-|c(x)|}/p(x)) \leq \log_2(\sum 2^{-|c(x)|})$ and by the Kraft-McMillan inequality, the latter value is at most 0.

2. Let $m_x = \lceil -\log_2 p(x) \rceil$. As $\sum_{x \in S} p(x) = 1$, we have $\sum_{x \in S} 2^{-m_x} \leq 1$, so by the Kraft-McMillan inequality, there is a prefix code c with code word lengths $\{m_x\}$. Now assign to each x the code word of length m_x in this code. We have $E[|c(x)|] = \sum p(x)|c(x)| = \sum p(x)m_x = \sum p(x)\lceil -\log_2 p(x) \rceil \leq \sum p(x)(-\log_2 p(x) + 1) = H[X] + 1$.

The code from part 2 of the proof of Theorem 8 prefix is called the McMillan code. By the theorem, it has an expected length which is at most 1 bit longer than than the optimum but it is not necessarily the optimum one. Note, however, that it has a stronger property than stated: Not only is the expected code length close to the entropy of the distribution, but the code length of each *individual* message is at most the self-entropy of that message plus one.

We next describe how to compute, from a tabulation of $\{p(x)\}_{x \in S}$ with $p(x) = \Pr[X = x]$ the prefix code c actually minimizing $E[|c(X)|]$. This code is called the *Huffman code*. For the Huffman code, it is more natural to think of prefix codes in terms of trees rather than dyadic intervals. We need the following lemma.

Lemma 9 *There is an optimum prefix code for X which assigns the two messages with the lowest probability (i.e., the two smallest value of $p(x)$) to two sibling leaves of maximum depth in the corresponding tree.*

Proof Consider an optimum prefix code for X . Let us first convince ourselves that there are two sibling leaves of maximum depth in the tree. Indeed, if a leaf l of maximum depth has no sibling, the code could be improved by removing l , thus turning its father into a leaf and assigning the message assigned to l to the father. Now, take any of the two siblings of maximum depth, if this leaf is assigned a message of probability q which is not one of the two smallest probabilities while some other leaf is assigned a message of probability $q' < q$, we may switch the two messages without increasing the value of $\sum p(x)|c(x)|$.

We can now describe the algorithm for finding the optimum tree given a probability vector (p_1, p_2, \dots, p_n) holding all values of $(p(x))_{x \in S}$.

1. If the vector contains only one value, return a leaf assigned this value.
2. Otherwise ensure that p_1 and p_2 are the smallest values - otherwise sort the vector.
3. Recursively find the optimum tree for the vector $(p_1 + p_2, p_3, \dots, p_n)$
4. Replace the leaf that was assigned $p_1 + p_2$ with a node with two sons, a leaf assigned p_1 and another leaf assigned p_2 .

We claim that the prefix code corresponding to the tree T found minimizes $\sum p(x)|c(x)|$, i.e., $\sum p_i \text{depth}(p_i)$. Denote this value by $v(T)$.

We show our claim by induction in the number of values in the probability vector. The claim is clearly true if there is only one value. Now suppose it fails for some larger input (p_1, \dots, p_n) but not for smaller values of n . Let T be the non-optimum tree found and let T' be the optimum tree for $(p_1 + p_2, \dots, p_n)$ found in the recursive call. Let T^* be an optimum tree for (p_1, \dots, p_n) . By the lemma, we can assume that p_1 and p_2 are assigned sibling leaves in T^* . Now let $T^{*'}$ be the tree where the father of p_1 and p_2 has been replaced with a single leaf holding $p_1 + p_2$. This is a tree for the vector $(p_1 + p_2, p_3, \dots, p_n)$. Clearly,

$$v(T) = v(T') + p_1 + p_2,$$

$$v(T^*) = v(T^{*'}) + p_1 + p_2,$$

so since $v(T) > v(T^*)$, we also have $v(T') > v(T^{*'})$, a contradiction.

The recursive procedure described may be time-consuming if implemented naively. However, it is easy to see that an iterative version of the procedure

can be implemented to run in time $O(n \log n)$ using standard data structures (in particular, a priority queue).

Since the Huffman code is an optimum code and the McMillan code satisfies $E[|c(X)|] < H[X] + 1$, so does the Huffman code. Furthermore, for the Huffman code, the following stronger bound can be shown:

$$E[|c(x)|] < H[X] + \min(1, p_{max} + 0.086)$$

where p_{max} is the probability of the most likely outcome of X .

4 Models and modelling

The general method of statistical coding is as follows: In order to construct a codec for a set of messages S , we first define a reasonable probability distribution over S (a *data model*) and then find a code c that makes $E[|c(x)|]$ small, ideally, as small as the entropy of the probability distribution (within one bit). Even stronger, we would like the message length of each individual code word to be close to the self-entropy of the corresponding message in the model.

Of course, the models we can handle are hardly ever a completely accurate reflection of the reality we wish to model. Rather, they are crude simplifications. We here list the most important classes of models that have proved useful for lossless compression of text and images. In all cases, we assume that our set of messages is Σ^n for a fixed alphabet Σ and a fixed length n (messages where the length is also variable and random can be handled at the cost of further complications)

iid or 0th order models: The text x to be encoded is assumed to be a sample of a random vector $X = (X_1, X_2, \dots, X_n)$ with the X_i mutually independent and identically distributed. The parameters of the model are the individual letter probabilities, i.e., $(p_\sigma)_{\sigma \in \Sigma}$, where $\forall i, \Pr[X_i = \sigma] = p_\sigma$.

k th order Markov models: Here, k is a positive integer. The text x to be encoded is assumed to be a sample of a random vector $X = (X_1, X_2, \dots, X_n)$ where, for all $i > k$ and all symbols $\sigma, \pi_1, \pi_2, \dots, \pi_k$,

$$\Pr[X_i = \sigma | X_{i-1} = \pi_1, X_{i-2} = \pi_2, \dots, X_{i-k} = \pi_k] = p_{\pi_k \pi_{k-1} \pi_{k-2} \dots \pi_1 \sigma},$$

where $(p_{\pi_k \pi_{k-1} \pi_{k-2} \dots \pi_1 \sigma})_{\pi_k, \pi_{k-1}, \pi_{k-2}, \dots, \pi_1, \sigma \in \Sigma}$ are the parameters of the model. That is, the probability distribution of a given letter in the text is determined completely by the previous k letters. The first k letters are to be considered parameters of the model.

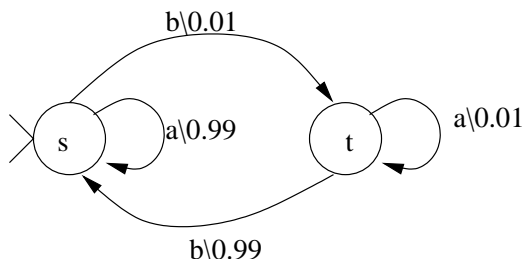


Figure 1: A two-state model

Finite state models: A finite state model is very similar to a finite automaton, but unlike a finite state automaton a finite state model defines a family of probability distributions rather than a formal language. Formally, a finite state model is a tuple $(Q, \Sigma, s, \delta, p)$ where Q is a finite set of states, Σ is the input alphabet, $s \in S$ is the start state of the model, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function and $p : Q \times \Sigma \rightarrow [0, 1]$ is a map so that for all $q \in Q$, $\sum_{\sigma \in \Sigma} p(q, \sigma) = 1$. Note that the entries Q, Σ, s and δ are exactly as in the definition of a finite automaton; only the map p is new. Finite state models may be represented graphically in a way very similar to finite state automata with the value of $p(q, \sigma)$ being written as an extra label on the arc between q and $\delta(q, \sigma)$. For instance, Figure 1 is the graphical representation of the model $(Q, \Sigma, s, \delta, p)$ with $Q = \{s, t\}, \Sigma = \{a, b\}, \delta(s, a) = s, \delta(s, b) = t, \delta(t, a) = t, \delta(t, b) = s, p(s, a) = p(t, b) = 0.99$ and $p(s, b) = p(t, a) = 0.01$.

For a given $n \geq 0$, a finite state model $(Q, \Sigma, s, \delta, p)$ defines a probability distribution on Σ^n defined by the following sampling procedure: We start in state $q_0 = s$ and choose the first symbol σ_1 in the string at random according to the probabilities $p(q_0, \sigma), \sigma \in \Sigma$. For example, in the model in Figure 1, we choose $\sigma_1 = a$ with probability 0.99 and $\sigma_1 = b$ with probability 0.01. Having chosen σ_1 in this way, we move to the state $q_1 = \delta(q_0, \sigma_1)$ and choose the next symbol σ_2 according to the probabilities $p(q_1, \sigma), \sigma \in \Sigma$ and then move to the state $q_2 = \delta(q_1, \sigma_2)$. We continue in this way until the entire string is defined.

It is easy to see that any family (one for each input length n) of probability distributions given by a k th order Markov model is also given by a finite state model (why?) but the converse is not the case. For instance, the model given in Figure 1 is not equivalent to any k th order Markov model, for any constant k . Thus, finite state models are more expressive than Markov models.

Hidden Markov models: Hidden Markov models are even more expressive than finite state models. To define a hidden Markov model, we first recall

that given two finite alphabets Σ_1 and Σ_2 , a *homomorphism* is given by an arbitrary map $f : \Sigma_1 \rightarrow \Sigma_2^*$ which is extended to $f : \Sigma_1^* \rightarrow \Sigma_2^*$ by the rules $f(\lambda) = \lambda$ and $\forall x, y \in \Sigma_1^*, f(x \cdot y) = f(x) \cdot f(y)$.

A hidden Markov model is given by a 1st order Markov model with set of symbols Σ_1 and a homomorphism $f : \Sigma_1 \rightarrow \Sigma_2^*$ and defines a probability distribution on Σ_2^n which may be sampled in the following way: We generate a sufficiently long string x using the 1st order Markov model, apply the homomorphism f and trim the resulting string $f(x)$ to length n (note that we may do this efficiently by applying the homomorphism symbol by symbol while generating the string x symbol by symbol).

These models are called hidden Markov models as we may think of them in the following way: Behind the scenes, a certain process generates the sequence of symbols in x , but we cannot observe these (they are “hidden”). Instead, when the process generates symbol σ , we observe $f(\sigma)$. Since $f(\sigma)$ may be the empty string, we may observe nothing when the process generates the symbol σ or we may see several symbols, if $|f(\sigma)| > 1$.

We leave as an exercise for the reader to prove that for any finite state model, there is an equivalent hidden Markov model. A hint: the set of symbols Σ_1 of the “hidden” process should be the set of possible transitions (i.e., arcs in the graphical representation) of the finite state model.

Linear Models: A Markov model of larger order provides, intuitively, a more refined model of a message than a Markov model of smaller order. We in general expect a more refined model to give a better compression than a less refined one. However, a disadvantage of the higher order models is the large number of parameters that describe the model. For some settings, such as encoding of a sound file, where each symbol of the alphabet is actually describing a (rounded) measurement on a linear scale, e.g., an integer in $\{0, 1, 2, \dots, m - 1\}$, a linear model becomes a meaningful special case of the Markov model. Again, we assume that each symbol depends on the previous k ones, but rather than the general situation outlined above, we assume

$$X_i = \text{round}(\alpha_1 X_{i-1} + \alpha_2 X_{i-2} + \dots + \alpha_k X_{i-k}) + Y_i, \text{ for } i \geq k + 1$$

where round is the function rounding a real value to the nearest integer and the Y_i 's are identically distributed and independent. Thus, the parameters of the model are $\alpha_1, \alpha_2, \dots, \alpha_k \in \mathbf{R}$ and $(p_\sigma)_{\sigma \in \Sigma}$ where $\forall i, \Pr[Y_i = \sigma] = p_\sigma$. Note that for a fixed sample x of $X = (X_1, X_2, \dots, X_n)$ (for instance, the sequence to be encoded) corresponds a fixed sample y of $Y = (Y_{k+1}, Y_2, \dots, Y_n)$. The sequence y is called the *residual* sequence of x .

Models for multidimensional Data: The Markov and linear models fit

one-dimensional text where the i 'th letter of the text is assumed to be somewhat predictable from the $(i - 1)$ 'st. For two dimensional data, such as images, the following variation makes more sense. Instead of viewing our data as a sample of a random vector $X \in \Sigma^n$, we view it as an $n \times m$ random matrix $X_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m$. We then assume that $X_{i,j}$ is somewhat predictable from entries close-by; the assumption analogous to the assumption of first order Markov models would be to assume

$$\Pr[X_{i,j} = \sigma | X_{i-1,j} = \pi_1, X_{i-1,j-1} = \pi_2, X_{i,j-1} = \pi_3] = p_{\pi_1 \pi_2 \pi_3 \sigma}$$

for all $i, j \geq 2$ and the analogous assumption to the linear model would be to assume

$$X_{i,j} = \text{round}(\alpha X_{i-1,j} + \beta X_{i-1,j-1} + \gamma X_{i,j-1}) + Y_{i,j}$$

for all $i, j \geq 2$ with the $Y_{i,j}$ s independent and identically distributed. Analogous definitions can be made for higher dimensional data.

Prediction models: All the models defined above share the following important property. There is an efficient algorithm, the *predictor*, that on input $(x_1, x_2, \dots, x_{i-1}) \in \Sigma^{i-1}$ outputs $(\Pr[X_i = \sigma | X_1 = x_1, X_2 = x_2, \dots, X_{i-1} = x_{i-1}])_{\sigma \in \Sigma}$, i.e., when a prefix of a sample of the distribution is given as input, the conditional probability distribution of the symbol following the prefix is given as output. This is easily seen for the Markov models, the finite state models and the linear models. For the hidden Markov models, it is also true, though the associated algorithm is more complicated and must use dynamic programming. The property also holds for the models for multidimensional data if we linearly order the symbols by, say, concatenating the rows of the matrix. It turns out, and we shall see in the next sections, that the prediction models are precisely those models for which we shall be able to construct efficient coders with code lengths matching the entropy bound. This also means that we have a high degree of freedom when choosing the model. Indeed, the focus on Markov and linear models above is primarily because these are the ones that are amenable to traditional statistical analysis. But, if we want to model, say, English text, we could have as our model any program (using, say, neural networks or other AI techniques) computing probabilities for i 'th symbol given the previous ones. Presumably, the more intelligence we build into the program the better a model of English text it is and hopefully makes the corresponding coder compress better.

Given a model, an interesting quantity to compute for a given message is the self-entropy of the message relative to the model, as this is the code length we expect a good coder based on the model to achieve for the message. Given

an algorithm implementing a predictor for a Prediction Model, we can easily compute the self-entropy of any message x in the following way, using the laws of conditional probability:

$$\begin{aligned}
 H(x) &= -\log_2 \Pr[X = x] \\
 &= -\log_2 \prod_{i=1}^{|x|} \Pr[X_i = x_i | X_{1..i-1} = x_{1..i-1}] \\
 &= \sum_{i=1}^{|x|} -\log_2 \Pr[X_i = x_i | X_{1..i-1} = x_{1..i-1}]
 \end{aligned}$$

5 Prediction models and Huffman Coding

Suppose our set of messages is Σ^n with an associated prediction model, given by an efficient algorithm A that on input $x_1x_2 \dots x_i \in \Sigma^i$ returns the conditional distribution of the $(i + 1)$ 'st symbol, given that the first symbols were $x_1x_2 \dots x_i$. Ideally, we want a codec matching the entropy bound. This would require building a Huffman tree over Σ^n which is clearly infeasible as soon as n becomes moderately large. Instead, we build for each $i \in \{1, \dots, n\}$ a Huffman tree over Σ using the probabilities given by the predictor for the i 'th symbol, given the preceding ones.

To be precise, the coder and decoder are described by the algorithms of Figures 2 and 3. Note that the decoder has access exactly to the parts of x it needs for calling the predictor A . Note also that it is essential that the coder and decoder use exactly the same program to derive a Huffman code from a given probability distribution.

Intuitively, since we apply a Huffman code to each symbol of the message rather than the message itself and this Huffman code may be at most one bit longer than the entropy of the conditional distribution of the symbol at that position, given the previous position, the expected length of the final code word, for a random sample from the model, should be at most n bit longer than the entropy of the model on Σ^n . Indeed, this can be shown to be true by a slightly involved calculation (which is much simpler for the special case of 0th order models or k th order Markov models). It is easy to see that we may indeed lose almost n bits compared to the optimal value. Consider the following simple 0th order model for $X = (X_1, \dots, X_{100}) \in \{0, 1\}^{100}$: $\Pr[X_i = 0] = 2^{-100}$, $\Pr[X_i = 1] = 1 - 2^{-100}$. The entropy of X is much smaller than 1 bit so the expected number of bits used to encode a message should

```

String Encode(String  $x$ )
  String  $y := \epsilon$ ;
  for  $i:=1$  to  $n$  do
    ProbabilityDistribution  $\mathbf{p} := A(x_1..x_{i-1})$ 
    PrefixCode  $c = \text{HuffmanCode}(\mathbf{p})$ 
     $y := y \cdot c(x_i)$ 
  od
return  $y$ 

```

Figure 2: Huffman coder using prediction model A .

```

String Decode(String  $y$ )
  String  $x := \epsilon$ ;
  for  $i:=1$  to  $n$  do
    ProbabilityDistribution  $\mathbf{p} := A(x_1..x_{i-1})$ 
    PrefixCode  $c = \text{HuffmanCode}(\mathbf{p})$ 
    Remove from  $y$  the prefix which is a  $c$ -codeword.
    Append to  $x$  the symbol this prefix is encoding.
  od
return  $x$ 

```

Figure 3: Huffman decoder using prediction model A .

be less than 2. Yet we use 100 bits to encode any message, as the Huffman code for any distribution on the binary alphabet is the trivial encoding (0 is encoded 0, 1 is encoded 1).

If a loss of n bits compared to optimal is unacceptable, we may improve the performance by coding symbols in blocks of k for some value $k > 1$, i.e., consider the message x to be a message in $(\Sigma^k)^{n/2}$ and modifying the predictor to be computing the probability distribution of the k next symbols. Our excess in expected length compared to the entropy would then be at most roughly n/k bits. Such a code is called a *Block code*.

We shall later see an alternative to using Huffman coding at symbol level. This alternative, *Arithmetic coding* encodes every message to a length equal to the self-entropy of the message in the model plus at most 2.

6 Parameter estimation and Model transmission

So far we have left out an important problem: How do we choose the parameters of our model? It may be that we are making a special purpose codec for, say, English text and in that case there are well-accepted 0th order and k th order models for small k that are generally accepted as describing English and hence we may use those.

It is, however, more likely that we want to estimate the parameters at compression time based on the message to be compressed to make sure that our model fits the characteristics of the message. Statistics has a traditional way of estimating parameters, namely, the method of *maximum likelihood estimation*. Maximum likelihood estimation prescribes that we should set the parameters of the model to the vector of values that maximizes the probability of the outcome x we actually observe. In the context of data compression we can argue that this principle of statistics (which is not in general completely uncontroversial) is sound: We expect to use the model to construct a coder which will encode our message to a length which is equal to (or at least close to) its self-entropy. But the self-entropy is just the logarithm of the reciprocal probability. Thus, by setting the parameters so that the probability of the outcome observed is *maximized*, we have also found the parameters *minimizing* its self-entropy - and hence the length of the encoded message.

When we make the parameters message-specific we have to somehow encode the parameters as part of the compressed data. The length of the encoding

of the parameters will be an extra overhead, not encountered for in our entropy estimate of the code length. This is why we in general prefer the number of parameters of our model to be short - it makes the overhead of transmitting the parameters of the model negligible.

For the 0th order model, the maximum likelihood parameters are the letter frequencies of the message x . That is, we set

$$p_\sigma = \frac{|\{i|x_i = \sigma\}|}{n}.$$

The entropy of the resulting model is also the self-entropy of the message in the model (exercise). This value is also called the *empiric 0th order entropy* of the string x and is usually² denoted $H_0(x)$.

Similarly, for the k th order Markov model, the maximum likelihood parameters are the conditional letter frequencies of the message. That is, we set $p_{\sigma\pi_1\dots\pi_k}$ to be the fraction of times σ occurred after $\pi_k\dots\pi_1$ among all the times $\pi_k\dots\pi_1$ occurred. This of course is only well-defined if $\pi_k\dots\pi_1$ occurred at all in the text. If not, we can set the value arbitrarily (it will not make any difference for the coder and decoder). The self-entropy of the string x in the resulting model is called the *k th order empiric entropy* of x and is usually³ denoted $H_k(x)$. Unlike the 0th order case, this value is not necessarily equal to the entropy of the model.

For the case of finite state model, if the entries Q, Σ, s and δ are fixed the maximum likelihood value of the map p for a given string $x \in \Sigma^n$ may be computed in the following way: We “run” the model on the string, observing the states

$$\begin{aligned} q_0 &= s \\ q_1 &= \delta(q_0, x_1) \\ &\dots \\ q_{n-1} &= \delta(q_0, x_{n-1}) \end{aligned}$$

the model enters at each time step if it indeed generates the string x . Then for each state $q \in Q$, we find the set $I_q = \{i|q_i = q\}$ and look at the symbols

²Kieffer uses the notation $H(x)$.

³Kieffer uses the notation $H_k(x)$ to mean something else, namely the 0th order entropy of the sequence obtained by bundling symbols in blocks of length k , i.e., $H_k^{\text{Kieffer}} = H_0((x_1, x_2, \dots, x_k), (x_{k+1}, \dots, x_{2k}), \dots)$. He uses the notation $H(x|k)$ for a quantity closely related to our $H_k(x)$. To be precise, $H(x|k) = \frac{1}{n}(H_0(x_1, x_2, \dots, x_k) + H_k(x))$. Thus, $H(x|k)$ normalizes the quantity $H_k(x)$ by dividing by the number of symbols and also takes the empiric 0th order entropy of the first k symbols into account.

emitted by the model at these points in time i.e., $x_{i+1}, i \in I_q$. The maximum likelihood value of $p(q, \sigma)$ for a $\sigma \in \Sigma$ is then the fraction of these symbols that were σ , i.e.,

$$p(q, \sigma) = \frac{|\{i \in I_q | x_{i+1} = \sigma\}|}{|I_q|}.$$

For the linear models, it is unfortunately not known how to compute the maximum likelihood parameters in an efficient way. Instead, the following technique can be used to find a reasonable set of parameters. We first try to choose the setting of the α_j parameters that minimizes the *variance* $\sum_{i=k+1}^n y_i^2$ of the residual sequence y . Note that if y has variance 0 it also has empiric 0th order entropy 0 and in general, a small variance of an integer valued sequence means that the sequence consists of small values; hence there may be only a few of them; yielding a small empiric 0th order entropy. Thus, we should find the coefficients α_i minimizing

$$h(\alpha_1, \alpha_2, \dots, \alpha_k) := \sum_{i=k+1}^n (x_i - \text{round}(\sum_{j=1}^k \alpha_j x_{i-k}))^2.$$

Unfortunately, because of the rounding, there is no known feasible way of performing this minimization either. So instead, we settle for finding the coefficients α_i minimizing

$$h(\alpha_1, \alpha_2, \dots, \alpha_k) := \sum_{i=k+1}^n (x_i - \sum_{j=1}^k \alpha_j x_{i-k})^2.$$

For a fixed sequence x , this is a quadratic form in the α_j which can be minimized efficiently by solving the system of linear equations

$$\frac{\partial h}{\partial \alpha_j} = 0, j = 1, \dots, k,$$

using Gaussian elimination. The *Levinson-Durbin algorithm* is a numerically stable algorithm for doing this calculation, taking advantage of the special structure of the quadratic form. Having found our estimate for the α -values we compute the residual sequence and do a 0th order estimate of the symbol probabilities of the y sequence, as above.

7 Adaptive models

Adaptive models form an attractive way of avoiding the problem of having to include a description of the parameters of the model as part of the code.

An adaptive model is a parameter-less prediction model that in a sense includes the process of parameter estimation as part of the model. As there are no parameters of the model, there are none to be coded!

An example is the following “adaptive 0th order model” (which is not a 0th order model at all, as the variables of the model are not independent):

$$\Pr[X_i = \sigma | X_{1..(i-1)} = x_{1..(i-1)}] = \frac{|\{j \in 1..i-1 | x_j = \sigma\}| + 1}{i-1 + |\Sigma|}.$$

Note that if the fraction $\frac{|\{j \in 1..i-1 | x_j = \sigma\}| + 1}{i-1 + |\Sigma|}$ was replaced with $\frac{|\{j \in 1..i-1 | x_j = \sigma\}|}{i-1}$ the model would simply define the probability of the symbol σ occurring in the i th position of the message as the frequency with which it occurred previously. Indeed, doing this is the basic idea of the model. The extra terms are added because of the *0-frequency problem* - we don't want a symbol to have probability 0 the first time it occurs as this would mean a self-entropy of ∞ and no finite length code assigned to it.

Adaptive k th order Markov models may be defined in a completely analogous way. In words, given a particular sequence $x_1 x_2 \dots x_{i-1}$ of preceding symbols, the model defines the probability of the i 'th symbol being σ as the frequency with which σ occurred after $x_{i-k} \dots x_{i-1}$ among all occurrences of $x_{i-k} \dots x_{i-1}$ in $x_1 x_2 \dots x_{i-1}$ - with this frequency slightly adjusted as to avoid the 0-frequency problem.

The adaptive models are not attractive only because of their parameterlessness. They may also perform better in terms of self-entropy (and hence compression) on many natural messages. Suppose we have a text whose characteristics varies as the text proceeds (perhaps it first has 100000 characters of English text, then followed by a 200000 character translation into French). Consider compressing it using a coder based on the following version of the 0th order adaptive model that does statistics only in a window of length 1000 into the past.

$$\Pr[X_i = \sigma | X_{1..(i-1)} = x_{1..(i-1)}] = \frac{|\{j \in \max(1, i-1000) \dots i-1 | x_j = \sigma\}| + 1}{\min(i-1, 1000) + |\Sigma|}.$$

Since the model should behave like an approximate 0th order model of English text while it compresses most of the English part and like an approximate 0th order model of French text while it compresses the most of the French part, the size of the compressed file is likely to be smaller than the size would have been had we used a non-adaptive 0th order model for the entire text.

8 Huffman coding for 0th and k th order models

The general framework above becomes somewhat simpler for the special cases of 0th order models and k th order Markov models.

For a 0th order model with maximum likelihood parameters for a text $x \in \Sigma^n$, the algorithms of Figures 2 and Figures 3 simplifies to simply encoding/decoding each letter of the text using a Huffman code for Σ based on the letter frequencies in x .

A coder that encodes each letter of a message from Σ^* according to a fixed prefix code for Σ is called a *memoryless code*. By restricting our attention to such codes, we can argue that the code we have constructed is good, without resorting to the framework of statistical coding theory.

Proposition 10 *The code just described compresses the text x to the minimum length possible among all memoryless codes.*

Proof Consider any memoryless code based on a prefix code c on Σ and let T be the tree corresponding to c . Let the frequency of the symbol σ in x be f_σ . The length of the code of x , $|x| = n$, is exactly $\sum_{i=1}^n |c(x_i)| = \sum_{\sigma \in \Sigma} n f_\sigma |c(\sigma)| = n v(T)$, where $v(T)$ is the weighted average depth of T , borrowing the terminology used in Section 3. As the Huffman tree minimizes $v(T)$ among all possible trees T , we have proved the proposition.

We have not yet accounted for encoding the parameters of the model. As the decoder just needs to know the Huffman tree, we can encode a description of this tree rather than encoding parameters of the model themselves, i.e., the letter frequencies. This is the more attractive alternative as the description of the tree can be made quite compact.

For a k th order Markov model with maximum likelihood parameters for a message $x \in \Sigma^n$, the algorithms of Figures 2 and Figures 3 simplifies to encoding/decoding each letter of the text using a Huffman code that depends on the setting of the previous k letters. A code that encodes each letter of a message from Σ^* according to a prefix code for Σ depending on the previous k symbols is called a *code of (finite) memory k* . As in Proposition 10, by restricting our attention to this special kind of codes, we can argue that the code we have constructed is good.

Proposition 11 *The code just described compresses the text x to the minimum length possible among all codes of memory k .*

We omit the proof which is very similar to the proof of Proposition 10. Note that in both propositions, the overhead associated with transmitting the model (or the Huffman trees) is not counted. For the k th order case, this overhead may be significant, as we have to describe $|\Sigma|^k$ different Huffman trees to specify the code.

9 Arithmetic coding

Arithmetic coding is a method that allows us to take any prediction model for a message space $S = \Sigma^n$ and encode any message $x \in S$ to a code word of length at most $H(x)+2$, where $H(x)$ is the self-entropy of x within the model. Thus, unlike the Huffman coder that may lose one bit *per symbol* compared to the optimum coding length, an arithmetic coder loses at most two bits *per message*, where a message may be, say, the Encyclopedia Britannica. Also, the coder is at least as time-efficient as the Huffman coder. The difference between code length and entropy is called the *redundancy* of the code (relative to the data model). Thus, arithmetic coding has a redundancy of at most two bits *per message*, while Huffman coding has a redundancy of at most one bit *per symbol*. The philosophy of statistical coding theory is a decoupling of modelling and coding. From this point of view, arithmetic coding is the end of the story as far as the coding part goes as we are not going to worry about two bits of redundancy for encoding an entire encyclopedia; all remaining effort thus can and should be spent on modelling. The main problem with arithmetic coding and the main reason why it is not used as often as Huffman coding in actual software is the fact that it is protected by a large number of patents.

The Shannon-Fano code

Arithmetic coding is based on (and can be seen as a special case of) the *Shannon-Fano code*, which may be described as follows, using the terminology of dyadic intervals. Given a set of messages S with an associated model and a *total order on S* , let the dyadic interval associated with $a \in S$ be the *longest dyadic interval* inside

$$[\Pr[X < a]; \Pr[X \leq a]),$$

where X is a random sample according to the model.

First note that $([\Pr[X < a]; \Pr[X \leq a]))_{a \in S}$ is a partition of $[0; 1)$, so the code defined is indeed a prefix code. Second, let us compare the code length

$c(a)$ to the self-entropy $H(a) = -\log_2 \Pr[X = a]$ of a . Note that the length of $[\Pr[X < a]; \Pr[X \leq a])$ is exactly $\Pr[X = a]$. Thus, the following lemma tells us that $|c(a)| \leq \lceil H(a) \rceil + 1$, i.e., $|c(a)| < H(a) + 2$.

Lemma 12 *Let $I = [t, t + \delta)$ be any subinterval of $[0; 1)$. The interval I contains, as a subset, a dyadic interval of length $2^{-(\lceil -\log \delta \rceil + 1)}$.*

Proof Let $k = \lceil -\log \delta \rceil + 1$. Note that $2^{-k} \leq \delta/2$. Thus, the first half of I contains an integer multiple of 2^{-k} , say $m2^{-k}$. Then, $[m2^{-k}; (m+1)2^{-k})$ is a dyadic interval which is a subset of I .

Note that the only way that the interval I may contain a larger dyadic interval than given by the lemma is by being a dyadic interval itself. In practice, we won't bother to potentially save a single bit by checking for this and will always just choose the interval specified in the proof of the lemma. Thus, the code word $c(a)$ is just the k -bit binary notation of the integer $\lceil \Pr[X < a]/2^{-k} \rceil$ where $k = \lceil -\log \Pr[X = a] \rceil + 1$.

Infinite precision arithmetic coding

The Shannon-Fano code assumes some ordering on S without specifying which. Infinite precision arithmetic coding is simply the Shannon-Fano code on Σ^n with the total order being the *lexicographic order*, assuming an arbitrary total order on Σ . The crucial observation is that if the model on Σ^n is a prediction model, the resulting infinite precision arithmetic code becomes computable by a (fairly) efficient algorithm. Indeed, we just need to compute $k = \lceil -\log \Pr[X = a] \rceil + 1$ and $\lceil \Pr[X < a]/2^{-k} \rceil$. We already know how to efficiently compute $H(a) = -\log \Pr[X = a]$ under a prediction model by iteratively computing $\Pr[X_{1..j} = a_{1..j}]$ for $j = 1, \dots, n$. Similarly, $\Pr[X < a]$ may be iteratively computed by noting that with the lexicographic order on Σ^n , we have

$$\begin{aligned} \Pr[X_{1..j} < a_{1..j}] &= \Pr[X_{1..j-1} < a_{1..j-1}] + \Pr[X_{1..j-1} = a_{1..j-1} \wedge X_j < a_j] \\ &= \Pr[X_{1..j-1} < a_{1..j-1}] + \Pr[X_j < a_j | X_{1..j-1} = a_{1..j-1}] \Pr[X_{1..j-1} = a_{1..j-1}] \end{aligned}$$

This leads to algorithm for computing the code in Figure 4, assuming infinite precision arithmetic. A similar algorithm can be made for decoding: We essentially simulate the encoding algorithm, repeatedly trying each possible value of a_j until the unique one consistent with the actual output is found. We omit the details.

```

String Encode(String  $a$ )
  Real low := 0;
  Real high := 1;
  Real range := high - low;
  for  $j:=1$  to  $n$  do
    ProbabilityDistribution  $\mathbf{p}$  :=  $A(a_1..a_{j-1})$ 
    low := low +  $(\sum_{k < a_j} \mathbf{p}(k)) * \text{range}$ ;
    high := low +  $\mathbf{p}(a_j) * \text{range}$ ;
    range := high - low;
    Invariant: [low;high) is  $[\text{Pr}[X_{1..j} < a_{1..j}], \text{Pr}[X_{1..j} \leq a_{1..j}]$ )
  od
  return Max dyadic interval in [low; high)

```

Figure 4: Infinite precision arithmetic coder using prediction model A .

Finite precision arithmetic coding

The problem with the algorithm in Figure 4 is that the required infinite precision arithmetic is, if not impossible, then at least computationally very expensive. If we assume that our prediction model returns probabilities which are finite precision rationals with some fixed precision (say, integer multipla of 2^{-32} , a precision which should be adequate for most practical purposes), the algorithm can be implemented, but the implementation will have time complexity $\Theta(n^2)$ and also use internal objects containing $\Theta(n)$ bits. Indeed, in the j 'th iteration of the loop we'll be adding numbers with $\Theta(j)$ digits. These resource requirements are unacceptable for practical compression software.

Suppose we try to avoid the problem by just doing approximate calculation using floating point numbers. What will obviously happen, even for fairly small values of n , is that the values low and high become equal because of rounding errors, completely destroying the algorithm (if it happens in the j 'th iteration of the for-loop on input a , all strings identical to a on the first j letters will receive the same codeword).

Finite precision arithmetic coding is merely a small collection of tricks designed for avoiding these catastrophic consequences of rounding errors. These essential ticks are the ones protected by various patents.

The tricks are as follows.

1. Keeping the variables low and high as two fixed precision floating point

numbers leads to disaster as they soon become very close to one another. Instead, whenever the values low and high share their most significant bit, we *scale* the values by simply removing this bit. Note that this bit is also part of the code word we eventually output, so when scaling, we output the bit we remove.

For example if the value of low at one point is 0.101011 (in binary) and the value of high is 0.110011, we output the bit “1”, replace low with 0.01011 and high with 0.10011. It is easy to see that the rest of the algorithm can be kept unchanged. Now, if we ignore the possibility that low becomes very close to the value $\frac{1}{2}$ from below and high becomes very close to the value $\frac{1}{2}$ from above (say low = 0.011111 and high = 0.100001), the scaling ensures that the low and high values are kept far apart.

2. The intention of scaling is to maintain the invariant “high - low $\geq 1/4$ ”. The simple version just mentioned doesn’t quite accomplish this because of the problem of values very close to $\frac{1}{2}$, but we shall deal with this in a minute. Now suppose that we have indeed maintained the invariant “high - low $\geq 1/4$ ” at one point in the algorithm using scaling. Still, very small values of probabilities returned by the predictor may lead to a rounding error putting high and low equal in the next iteration. To ensure that this is not the case we control the rounding ourselves by keeping low and high as unsigned integers, rather than as floating point numbers. Specifically, we let low and high-1 be values in $0, \dots, 2^b - 1$ and interpret the integer value i as $i/2^{-b}$. Now, when implementing the arithmetic in the body of the for-loop, we implement the rounding making sure that for each possible value of a_j there is an interval of [low;high) of non-zero length. This can be done in various ad hoc ways and is trivial if we know that the predictor only outputs probability values $\gg 2^{-b}$: In that case, we just round to the nearest integer multiple of 2^{-b} . Note that this means that the code we are constructing is actually *not* equal to the infinite precision arithmetic code on the probabilities given by the predictor, rather, it is the infinite precision arithmetic code according to some slightly adjusted probabilities. However, again, as long as the probabilities given by the predictor are not too small, the extra redundancy of the code we suffer as a consequence of this is negligible.
3. We now only have to explain how to maintain the invariant “high - low $\geq 1/4$ ”. As explained above, the scaling procedure we have described does alone not accomplish this because of the possibility of both high

and low being very close to $1/2$. Dealing with this is the most clever trick so far. Suppose indeed that we are in the bad case:

- (a) We have $\text{high} - \text{low} < 1/4$, breaking the invariant making our rounding procedure safe.
- (b) We can't scale, as we neither have that both high and low are below $\frac{1}{2}$ or that they are both at least $\frac{1}{2}$.

Then it must be the case that both high and low are strictly between $\frac{1}{4}$ and $\frac{3}{4}$. *We then scale anyway*, by assigning low the value $(\text{low} - \frac{1}{4}) * 2$ and high the value $(\text{high} - \frac{1}{4}) * 2$. We don't know if the bit we should output is a 0 or 1 yet, as we don't know on which side of $\frac{1}{2}$ we will eventually fall. But note that our scaling has kept the value $\frac{1}{2}$ as a fixpoint. We'll refer to this kind of scaling as a non-standard scaling.

When we continue the algorithm it may be the case that in the next iteration, both high and low are below $\frac{1}{2}$. In that case, we scale as usual but output 01 rather than 0. Similarly, if they are both above $\frac{1}{2}$, we scale as in the standard way, but output 10 rather than 1. It may also be the case that we cannot scale and that we eventually again have $\text{high} - \text{low} < 1/4$. We then again do a non-standard scaling $\text{low} = (\text{low} - \frac{1}{4}) * 2$ and $\text{high} = (\text{high} - \frac{1}{4}) * 2$.

When we continue the algorithm it may be the case that in the next iteration, both high and low are below $\frac{1}{2}$. In that case, we scale as usual but output 011 rather than 0. Similarly, if they are both above $\frac{1}{2}$, we scale in the standard way, but output 100 rather than 1. It may also be the case that we cannot scale and that we eventually again have $\text{high} - \text{low} < 1/4$, and so on and so forth.

In general, we just have to remember the number of times k we made a non-standard scaling and if eventually both high and low are below $\frac{1}{2}$, we scale in the standard way and output 01^k . If eventually both high and low are above $\frac{1}{2}$, we scale in the standard way and output 10^k . In both cases, we are back to the normal mode of operation.

Finally, it may be the case the we terminate the algorithm with the last scalings we have done being non-standard. We then need to take some special care to output the right value in the end. We leave this as an exercise.

This concludes the description of finite precision arithmetic coding. As in the infinite precision case, the decoder works by simulating the encoder and in each iteration of the for loop selecting the unique value of a_i that is consistent

with the code word to be decoded and hence reconstructing the original message.