Analysis of Arithmetic Coding for Data Compression

Paul G. Howard and *Jeffrey Scott Vitter*

Technical Report No. CS–91–03
January 1991

Analysis of Arithmetic Coding for Data Compression

Paul G. Howard[1] and Jeffrey Scott Vitter[2]

Department of Computer Science
Brown University
Providence, R. I. 02912–1910

[1]Support was provided in part by NASA Graduate Student Researchers Program grant NGT–50420 and by an NSF Presidential Young Investigators Award with matching funds from IBM. Additional support was provided by a Universities Space Research Association/CESDIS appointment.

[2]Support was provided in part by an NSF Presidential Young Investigator Award with matching funds from IBM, by NSF grant CCR-9007851, and by ONR grant N00014–83–K–0146, ARPA Order No. 4786.

Abstract

Arithmetic coding, in conjunction with a suitable probabilistic model, can provide nearly optimal data compression. In this paper we analyze the effect that the model and the particular implementation of arithmetic coding have on the code length obtained. We show that adaptive models give the same code length as semi-adaptive decrementing models. Periodic scaling is often used in arithmetic coding implementations to reduce time and storage requirements; it also introduces a recency effect which can further affect compression. We introduce the concept of *weighted entropy* and use it to characterize in an elegant way the effect that periodic scaling has on the code length. We also give a rigorous proof that the coding effects of rounding scaled weights, using integer arithmetic, and encoding end-of-file are negligible.

Index terms: data compression, arithmetic coding, analysis of algorithms, adaptive modeling

1 Introduction

We analyze the amount of compression possible when arithmetic coding is used for text compression in conjunction with various input models. Arithmetic coding is a technique for statistical lossless encoding. It can be thought of as a generalization of Huffman coding [1] in which probabilities are not constrained to be integral powers of 2, and code lengths need not be integers.

The basic algorithm for encoding using arithmetic coding works as follows:

1. We begin with a "current interval" initialized to [0..1].
2. For each symbol of the file, we do two steps:
   (a) We subdivide the current interval into subintervals, one for each possible alphabet symbol. The size of a symbol's subinterval is proportional to the estimated probability that the symbol will be the next symbol in the file, according to the input model.
   (b) We select the subinterval corresponding to the symbol that actually occurs next in the file, and make it the new current interval.
3. We transmit enough bits to distinguish the final current interval from all other possible final intervals.

The length of the final subinterval is clearly equal to the product of the probabilities of the individual symbols, which is the probability p of the particular sequence of symbols in the file. The final step uses almost exactly $-\lg p$ bits to distinguish the file from all other possible files. For detailed descriptions of arithmetic coding, see [2] and especially [3].

Example 1: We illustrate a non-adaptive code, encoding the 8-symbol file *"abacbcba"* with arbitrary fixed probability estimates $p_a = 0.5$, $p_b = 0.3$, and $p_c = 0.2$. Encoding proceeds as follows:

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000	1.000000	0.5	0.3	0.2
a	0.000000	0.500000	0.5	0.3	0.2
b	0.250000	0.400000	0.5	0.3	0.2
a	0.250000	0.325000	0.5	0.3	0.2
c	0.310000	0.325000	0.5	0.3	0.2
b	0.317500	0.322000	0.5	0.3	0.2
c	0.321100	0.322000	0.5	0.3	0.2
b	0.321550	0.321820	0.5	0.3	0.2
a	0.321550	0.321685	0.5	0.3	0.2

In binary the final interval is [0.01010 01001 01001, 0.01010 01001 01101]. We can uniquely identify this 8-symbol file by transmitting "01010 01001 0101'. According to the fixed model, the probability p of this file is $(0.5)^3(0.3)^3(0.2)^2 = 0.000135$ (exactly the size of the final interval) and the code length (in bits) is $-\lg p = 12.855$. In practice we have to transmit 14 bits. (In this and all other examples we neglect the cost of designating the end of the file.) □

The coder in an arithmetic coding system must work in conjunction with a model that produces probability information, describing a file as a sequence of decisions; at each decision point it estimates the probability of each possible choice. The coder then uses the set of probabilities to encode the actual decision.

To ensure decodability, the model may use only information known by both the encoder and the decoder. Otherwise there are no restrictions on the model; in particular, it can change as the file is being encoded. In Section 2 we describe several typical models for context-independent text compression. We examine semi-adaptive models (those that use a preliminary pass to determine and encode exact symbol counts), and we introduce a new decrementing semi-adaptive model, proving that it improves upon the usual static semi-adaptive model. We describe adaptive models (those that implicitly "learn" symbol probabilities) and prove that one particular adaptive model gives exactly the same code length as our new decrementing semi-adaptive model.

Section 3 contains our main result, which precisely and provably characterizes the code length of a file dynamically coded with periodic count-scaling. We express the code length in terms of weighted entropies of the model, which are the entropies implied by the model at the scaling times. Our result shows both the advantage to be gained by scaling because of a locality-of-reference effect and the overhead loss incurred by the scaling process. In Section 4 we extend this analysis to higher-order models based on the partial string matching algorithm of Cleary and Witten.

Through the years practical adjustments have been made to arithmetic coding Lspace [3,4,5,6] to allow the use of integer rather than rational or floating point arithmetic and to transmit output bits almost in real time instead of all at the end. In Section 5 we prove that the loss of coding efficiency caused by practical coding requirements is negligible, thus demonstrating rigorously the empirical claims made in [3].

We use the following notation throughout this article:

$$
\begin{aligned}
t &= \text{length of the file, in bytes;}\\
n &= \text{number of symbols in the input alphabet;}\\
k &= \text{number of different alphabet symbols that occur in the file;}\\
a_i &= \text{the } i\text{th symbol in the alphabet;}\\
c_i &= \text{number of occurrences of } a_i \text{ in the file;}\\
p_i &= c_i/t \quad \text{(the probability of symbol } a_i \text{ in the file);}\\
\lg x &= \log_2 x;
\end{aligned}
$$

$$\begin{aligned} H(X) &= \textstyle\sum_i -x_i \lg x_i \quad \text{(the entropy implied by distribution } X\text{);} \\ A^{\overline{B}} &= A(A+1)\cdots(A+B-1) \quad \text{(the rising factorial function).} \end{aligned}$$

In this paper, results for a "typical file" refer to a file with $t = 100{,}000$, $n = 256$, and $k = 100$. Code lengths are expressed in bits. We assume 8-bit bytes.

2 Modeling Effects

The length of the encoded file depends on the model, and how it is used. Most models for text compression involve estimating the probability p of a symbol by

$$p = \frac{\text{weight of symbol}}{\text{total weight of all symbols}},$$

which we can then encode in $-\lg p$ bits using exact arithmetic coding. The probability estimation can be done *adaptively* (dynamically estimating the probability of each symbol based on all symbols that precede it), *semi-adaptively* (using a preliminary symbol count for the entire file), or *non-adaptively* (using fixed probabilities for all files). Non-adaptive models are not very interesting, since their effectiveness depends only on how well their probabilities happen to match the statistics of the file being encoded; the match can be arbitrarily bad [7].

In this section we show the relationship between semi-adaptive and adaptive models based on a zero-order Markov model, that is, context-independent symbol counts.

2.1 Semi-adaptive codes

Semi-adaptive codes are conceptually simple, and useful when real-time operation is not required; their main drawback is that they require knowledge of the file statistics prior to encoding. The statistics are normally used in a *static code*; that is, the probabilities used to encode the file remain the same during the entire encoding. We introduce *decrementing codes* and show that they improve on static codes.

2.1.1 Static and decrementing semi-adaptive codes

Assuming that encoding uses exact arithmetic, and that there are no computational artifacts, we can use the file statistics to form a static probabilistic model. Not including the cost of transmitting the model, the code length L_{SS} is

$$\begin{aligned} L_{SS} &= -\lg \prod_{i=1}^{n} (c_i/t)^{c_i} \\ &= t \lg t - \sum_{i=1}^{n} c_i \lg c_i, \end{aligned}$$

the information content of the file. Dividing by the file length gives the self-entropy of the file, and forms the basis for claims that arithmetic coding encodes asymptotically close to the entropy. What these claims really mean is that arithmetic coding can encode close to the entropy of a probabilistic model whose symbol probabilities are the same as those of the file, because computational effects (discussed in Section 5) are insignificant.

Example 2: We illustrate a static code with the same file as in Example 1. Based on exact symbol counts within the file, we have probabilities $p_a = 3/8$, $p_b = 3/8$, and $p_c = 1/4$, which we use during the entire encoding.

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000000	1.000000000	$3/8$	$3/8$	$2/8$
a	0.000000000	0.375000000	$3/8$	$3/8$	$2/8$
b	0.140625000	0.281250000	$3/8$	$3/8$	$2/8$
a	0.140625000	0.193359375	$3/8$	$3/8$	$2/8$
c	0.180175781	0.193359375	$3/8$	$3/8$	$2/8$
b	0.185119629	0.190063477	$3/8$	$3/8$	$2/8$
c	0.188827515	0.190063477	$3/8$	$3/8$	$2/8$
b	0.189291000	0.189754486	$3/8$	$3/8$	$2/8$
a	0.189291000	0.189464808	$3/8$	$3/8$	$2/8$

The final interval is [0.00110 00001 11011, 0.00110 00010 00000] in binary. The theoretical code length is $-\lg 0.000173808 = 12.490$ bits, slightly shorter than in Example 1, since the probabilities, instead of being arbitrary, reflect the actual frequencies of the symbols in the file. The actual code length is 13 bits, since the final subinterval can be determined from the output "00110 00001 111." □

If we know a file's exact statistics ahead of time, we can get even more compression by using a *decrementing code*. We modify the symbol weights dynamically (by decrementing each symbol's weight each time it is encoded) to reflect symbol frequencies for just that part of the file not yet encoded; hence for each symbol we always use the best available estimates of the next-symbol probabilities. The resulting code length is

$$L_{SD} = -\lg\left(\left(\prod_{i=1}^{k} c_i!\right)\Big/ t!\right). \qquad (1)$$

Example 3: We illustrate a decrementing code with the same file. Again we start with probabilities $p_a = 3/8$, $p_b = 3/8$, and $p_c = 1/4$, but during encoding the probabilities change.

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000000	1.000000000	$3/8$	$3/8$	$2/8$
a	0.000000000	0.375000000	$2/7$	$3/7$	$2/7$
b	0.107142857	0.267857143	$2/6$	$2/6$	$2/6$
a	0.107142857	0.160714286	$1/5$	$2/5$	$2/5$
c	0.139285714	0.160714286	$1/4$	$2/4$	$1/4$
b	0.144642857	0.155357143	$1/3$	$1/3$	$1/3$
c	0.151785714	0.155357143	$1/2$	$1/2$	$0/2$
b	0.153571429	0.155357143	$1/1$	$0/1$	$0/1$
a	0.153571429	0.155357143	$0/0$	$0/0$	$0/0$

The final interval is [0.00100 11101 01000, 0.00100 11111 00011] in binary. The theoretical code length is $-\lg 0.001785714 = 9.129$ bits, considerably shorter than in Example 2, because the probabilities used at each point reflect the actual frequencies of the symbols in just the remaining portion of the file, not the entire file. The actual code length is 10 bits, since the final subinterval can be determined from the output "00100 11110." □

In the following theorem we show that a decrementing code improves on the corresponding semi-adaptive static code:

Theorem 1 *For all input files, the code length of a semi-adaptive decrementing code is at most that of a semi-adaptive static code. Equality holds only when the file consists of repeated occurrences of a single letter.*

Proof: Neither L_{SS} nor L_{SD} depends on the order in which the symbols appear in the file, so we can inductively build up a file with the same code length as any given file by starting with one occurrence of each symbol, then adding the remaining occurrences of each symbol. We show that the inequality holds for files with one occurrence of each symbol, and that adding occurrences of any symbol maintains the inequality.

Using a superscript "1" to denote a file consisting of exactly one occurrence of each of k symbols, we see that $L^1_{SS} = -\lg(1/k^k)$ and $L^1_{SD} = -\lg(1/k!)$. Since $k^k > k!$ for all $k \geq 2$, we have $L^1_{SS} > L^1_{SD}$ for $k \geq 2$. If we have a file with t symbols in it and we add another occurrence of a symbol that already occurs c times, we increase L_{SS} by $\lg((t+1)/t)^t - \lg((c+1)/c)^c + \lg((t+1)/(c+1))$, and we increase L_{SD} by $\lg((t+1)/(c+1))$. Since $t > c$ and $\lg((x+1)/x)^x$ is an increasing function for $x > 0$, we have $\lg((t+1)/t)^t > \lg((c+1)/c)^c$, so L_{SS} increases by more than L_{SD}.

Finally, if $k = 1$ (the file consists of repeated occurrences of a single letter), $L_{SS} = L_{SD} = 0$. □

The idea of a decrementing code seems to be new. Static codes have been widely used in conjunction with Huffman coding, where they are appropriate since changing weights can require changing the structure of the coding tree. Our theorem does not contradict Shannon's theorem [8]; he discusses only the best static code.

2.1.2 Encoding the model

If we assume that all symbol distributions are equally likely for a given file length t, the cost of transmitting the exact model statistics is

$$\begin{aligned} L_M &= \lg(\text{number of possible distributions}) \\ &= \lg \binom{t+n-1}{n-1} \\ &\sim n \lg(et/n). \end{aligned} \tag{2}$$

Example 4: For our short sample file, $t = 8$ and $n = 3$, so $L_M = \lg \binom{10}{2} = \lg 45 = 5.492$. □

A similar result using enumerative codes appears in [7] and [9]. For a typical file L_M is only about 2560 bits, or 320 bytes. The assumption of equally-likely distributions is not very good for text files; in practice we can reduce the cost of encoding the model by 50 percent or more by encoding each of the counts using a suitable encoding of the integers, such as Fibonacci coding [10].

Strictly speaking we must also encode the file length t before encoding the model; the cost is insignificant, between $\lg t$ and $2 \lg t$ bits using an appropriate encoding of integers [11,12,13].

2.2 Adaptive codes

Adaptive codes use a continually changing model, in which we use the frequency of each symbol up to a given point in the file as an estimate of its probability at that point.

We can encode a symbol using arithmetic coding only if its probability is nonzero, but in an adaptive code we have no way of estimating the probability of a symbol before it has occurred for the first time. This is the *zero-frequency problem*, discussed in detail in [7] and [14]. For large files with small alphabets and simple models, all solutions to this problem give roughly the same compression. In this section we adopt the solution used in [3], simply assigning an initial weight of 1 to all alphabet symbols. The code length using this adaptive code is

$$L_A = -\lg \left(\left(\prod_{i=1}^{k} c_i! \right) \Big/ n^{\bar{t}} \right). \tag{3}$$

Example 5: Once again we encode the same file, starting with counts $c_a = 1$, $c_b = 1$, and $c_c = 1$.

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000000	1.000000000	$1/3$	$1/3$	$1/3$
a	0.000000000	0.333333333	$2/4$	$1/4$	$1/4$
b	0.166666667	0.250000000	$2/5$	$2/5$	$1/5$
a	0.166666667	0.200000000	$3/6$	$2/6$	$1/6$
c	0.194444444	0.200000000	$3/7$	$2/7$	$2/7$
b	0.196825397	0.198412698	$3/8$	$3/8$	$2/8$
c	0.198015873	0.198412698	$3/9$	$3/9$	$3/9$
b	0.198148148	0.198280423	$3/10$	$4/10$	$3/10$
a	0.198148148	0.198187831	$4/11$	$4/11$	$3/11$

The final interval is [0.00110 01010 11100 11101, 0.00110 01010 11110 00111] in binary. The theoretical code length is $-\lg 0.000039683 = 14.621$ bits, 5.492 bits longer than in Example 3; the difference is exactly the cost of sending the model that we found in Example 4. The actual code length is 15 bits, since the final subinterval can be determined from the output "00110 01010 11101." □

We show the equivalence of adaptive and semi-adaptive codes in the following theorem:

Theorem 2 *For all input files, the adaptive code with initial 1-weights gives the same code length as the semi-adaptive decrementing code in which the input model is encoded based on the assumption that all symbol distributions are equally likely. In other words,* $L_A = L_{SD} + L_M$.

Proof: Combine equations (1), (2), and (3), noting that $\binom{t+n-1}{n-1} = n^{\bar{t}}/t!$. □

Our result improves on that of Cleary, Witten, and Bell [7,9] in that it applies to practical codes and shows exact, not approximate, equality. The intuition is that in the adaptive code, the cost of "learning" the model is not avoided, but merely spread over the entire file.

3 Scaling

Scaling is the process in which we periodically reduce the weights of all symbols. It allows us to use lower precision arithmetic at the expense of making the model more approximate, which can hurt compression. It also introduces a *locality of reference* (or *recency*) effect, which often improves compression. In this section we give a precise characterization of the effect of scaling on code length, in terms of an elegant notion we introduce called *weighted entropy.*

In most text files we find that most of the occurrences of at least some words are clustered in one part of the file. We can take advantage of this locality by assigning more weight to recent occurrences of a symbol in a dynamic model. In practice there are several ways to do this:

- Periodically restarting the model. This usually discards too much information to be effective.

- Using a sliding window on the text [15]. This requires excessive computational resources.

- Recency rank coding [16,17,18]. This is simple but corresponds to a rather coarse model of recency.

- Exponential aging (giving exponentially increasing weights to successive symbols) [19,20]. This is moderately difficult to implement because of the changing weight increments.

- Periodic scaling [3]. This is simple to implement, fast and effective in operation, and amenable to analysis. It also has the computationally desirable property of keeping the symbol weights small. In effect, scaling is a practical version of exponential aging. This is the method that we analyze.

In our discussion of scaling, we assume that a file is divided into *blocks* of length B. Our model assumes that at the end of each block, the weights for all symbols are multiplied by a scaling factor f, usually $1/2$. Within a block we update the symbol weights by adding 1 for each occurrence.

Example 6: In this example, we set the scaling threshold at 10, so we scale only once, just before the last symbol in the file. To retain integer weights without allowing any weight to fall to 0, we round all fractional weights obtained during scaling to the next higher integer.

Symbol	Current Interval		p_a	p_b	p_c
Start	0.000000000	1.000000000	$1/3$	$1/3$	$1/3$
a	0.000000000	0.333333333	$2/4$	$1/4$	$1/4$
b	0.166666667	0.250000000	$2/5$	$2/5$	$1/5$
a	0.166666667	0.200000000	$3/6$	$2/6$	$1/6$
c	0.194444444	0.200000000	$3/7$	$2/7$	$2/7$
b	0.196825397	0.198412698	$3/8$	$3/8$	$2/8$
c	0.198015873	0.198412698	$3/9$	$3/9$	$3/9$
b	0.198148148	0.198280423	$3/10$	$4/10$	$3/10$
Scaling			$2/6$	$2/6$	$2/6$
a	0.198148148	0.198192240	$3/7$	$2/7$	$2/7$

The final interval is [0.00110 01010 11100 11101, 0.00110 01010 11110 01011] in binary. The theoretical code length is $-\lg 0.000044092 = 14.469$ bits, slightly shorter than in Example 5. The actual code length is 15 bits, since the final subinterval can be determined from the output "00110 01010 11101." □

In this section we assume exact rational arithmetic. We introduce the following additional notation for the analysis of scaling:

$$\begin{aligned}
s_i &= \text{weight of symbol } a_i \text{ at the start of the block;} \\
c_i &= \text{number of occurrences of symbol } a_i \text{ in the block;} \\
m &= \text{the minimum weight allowed for any symbol, usually 1;} \\
A &= \sum_{i=1}^{n} s_i \quad \text{(the total weight at the start of the block);} \\
B &= \sum_{i=1}^{n} c_i \quad \text{(the size of the block);} \\
C &= A + B \quad \text{(the total weight at the end of the block);} \\
f &= A/C \quad \text{(the scaling factor);} \\
q_i &= s_i/A \quad \text{(probability of symbol } a_i \text{ at the start of the block);} \\
p_i &= c_i/B \quad \text{(probability of symbol } a_i \text{ in the block);} \\
r_i &= (s_i + c_i)/C \quad \text{(probability of symbol } a_i \text{ at the end of the block);} \\
Q, R &= \text{probability distributions } \{q_i\} \text{ and } \{r_i\}; \\
b &= \text{number of blocks resulting from scaling.}
\end{aligned}$$

When $f = 1/2$, we scale by halving all weights every B symbols, so $A = B$ and $b \approx t/B$. In a typical implementation, $A = B = 8192$.

3.1 Coding Theorem for Scaling

In our scaling model each symbol's counts are multiplied by f whenever scaling takes place. Thus scaling has the effect of giving more weight to more recent symbols. If we denote the number of occurrences of symbol a_i in block m by $c_{i,m}$, the weight $w_{i,m}$ of symbol a_i after m blocks is given by

$$\begin{aligned}
w_{i,m} &= c_{i,m} + f w_{i,m-1} \\
&= c_{i,m} + f c_{i,m-1} + f^2 c_{i,m-2} + \cdots.
\end{aligned}$$

The weighted probability $p_{i,m}$ of a_i at the end of block m is then

$$p_{i,m} = \frac{w_{i,m}}{\sum_{i=1}^{n} w_{i,m}}.$$

We now define a weighted entropy in terms of the weighted probabilities:

Definition 1 The *weighted entropy* of a file at the end of the mth block, denoted by H_m, is the entropy implied by the probability distribution at that time, computed according to the scaling model. That is,

$$H_m = \sum_{i=1}^{n} -p_{i,m} \lg p_{i,m}.$$

We find that l_m, the average code length per symbol for block m, is related to the starting weighted entropy H_{m-1} and the ending weighted entropy H_m in a particularly simple way:

$$\begin{aligned} l_m &\approx \frac{1}{1-f} H_m - \frac{f}{1-f} H_{m-1} \\ &= H_m + \frac{f}{1-f}(H_m - H_{m-1}). \end{aligned}$$

Letting $f = 1/2$, we obtain

$$l_m \approx 2H_m - H_{m-1}.$$

When we multiply by the block length and sum over all blocks, we obtain the following precise and elegant characterization of the code length in terms of weighted entropies:

Theorem 3 *Let L be the compressed length of a file. Then*

$$\begin{aligned} & B\left(\left(\sum_{m=1}^{b} H_m\right) + H_b - H_0\right) - O(kt) \\ & \qquad \le L \le B\left(\left(\sum_{m=1}^{b} H_m\right) + H_b - H_0\right) + O(kt), \end{aligned}$$

where $H_0 = \lg n$ is the entropy of the initial model and H_m is the (weighted) entropy implied by the scaling model's probability distribution at the end of block m. The constants in the $O(kt)$ terms are small, namely $1/B$ in the lower bound and less than $(\lg B)/B$ in the upper bound.

3.1.1 Proof of the upper bound

We prove the upper bound of Theorem 3 by showing first that the code length of a block depends only on the beginning weights and block counts of the symbols, and not on the order of symbols within the block. Then we show that there is an order such that for each symbol certain equalities and inequalities hold, which we use to compute a value and an upper bound for the code length of all occurrences of a single symbol in one block. Finally we sum over all symbols to obtain the worst case code length of a block, and over all blocks to obtain the code length of the file. The proof of the lower bound is similar.

The first lemma enables us to choose any convenient symbol order within a block.

Lemma 1 *The code length of a block depends only on the beginning weights and block counts of the symbols, and not on the order of symbols within the block.*

Proof: The exact code length L of a block,

$$L = -\lg\left(\left(\prod_{i=1}^{k} s_i^{\overline{c_i}}\right) \Big/ A^{\overline{B}}\right),$$

has no dependence on the order of symbols. For each symbol a_i the numerators in the set of probabilities used for coding the symbol in the block always form the sequence $\langle s_i, s_i+1, \ldots, s_i+c_i-1\rangle$, and for the block as a whole the denominators always form the sequence $\langle A, A+1, \ldots, A+B-1\rangle$. □

In the next lemma we prove the almost obvious fact that there is some order of symbols such that the occurrences of each symbol are roughly evenly distributed through the block. This order will enable us to use a smoothly varying function to estimate the probabilities used for coding the occurrences.

Definition 2 A block of length B containing $c_1, c_2, \ldots, c_k$ occurrences of symbols $a_1, a_2, \ldots, a_k$, respectively, has the *evenly-distributed* (or *ED*) property if for each symbol a_i and for all m, $1 \le m \le c_i$, the symbol occurs for the mth time not after position mB/c_i.

Lemma 2 *Every distribution of symbol counts has an order with the ED property.*

Proof: Let $r_i(j)$ be the number of occurrences of symbol a_i required up through the jth position in any ED order. Such an order exists if and only if $\sum_{i=1}^{k} r_i(j) \le j$ for $1 \le j \le B$. We find that $r_i(j) = \lceil \frac{c_i(j+1)}{B} \rceil - 1 < c_i(j+1)/B$. Since $\sum_{i=1}^{k} c_i = B$, $\sum_{i=1}^{k} r_i(j) < j+1$, or $\sum_{i=1}^{k} r_i(j) \le j$ since $\sum_{i=1}^{k} r_i(j)$ is an integer. This holds for any j, so an ED order exists. □

Next we define a number of related sums and integrals approximating l, the code length of one symbol in one block. For a given symbol with probability $p = c/B$ within the block, we can divide the block into c subblocks each of length $B/c = 1/p$. Then $p_B(k)$ and $p_E(k)$ give the probability that the dynamic model would give to the kth occurrence of the symbol if it occurred precisely at the beginning and end of the kth subblock respectively.

$$p_B(k) = \frac{qA+k}{A+k/p}; \qquad p_E(k) = \frac{qA+k}{A+(k+1)/p}.$$

The expressions S_L and S_U are the lower and upper bounds of the symbol's code length, based on its occurrences being as early or as late as possible in the subblocks.

$$S_L = \sum_{j=0}^{c-1} -\lg p_B(j); \qquad S_U = \sum_{j=0}^{c-1} -\lg p_E(j).$$

The integrals I_L and I_U approximate S_L and S_U.

$$I_L = \int_0^c -\lg p_B(x)\,dx; \qquad I_U = \int_0^c -\lg p_E(x)\,dx.$$

We define Δ_I to be $I_U - I_L$. After a considerable amount of algebra, we get

$$\begin{aligned}\Delta_I &= \frac{1}{1-f}((c+1-f)\lg(c+1-f) - c\lg c \\ &\qquad -(fc+1-f)\lg(fc+1-f) + fc\lg(fc)).\end{aligned}$$

The expressions Δ_U and Δ_L are used to bound the error in approximating S_U by I_U and S_L by I_L respectively.

$$\Delta_U = \begin{cases} \lg(1+\frac{c}{qA}) - \lg(1+\frac{c}{pA+1}) & \text{if } p > q - 1/A; \\ 0 & \text{if } p \le q - 1/A; \end{cases}$$

$$\Delta_L = \begin{cases} \lg(1+\frac{c}{pA}) - \lg(1+\frac{c}{qA}) & \text{if } p < q; \\ 0 & \text{if } p \ge q. \end{cases}$$

We need a simple lemma from integral calculus.

Lemma 3 *If $g(x)$ is monotone increasing, then*

$$\sum_{j=0}^{c-1} g(j) < \int_0^c g(x)\,dx.$$

If $g(x)$ is monotone decreasing, then

$$\sum_{j=0}^{c-1} g(j) < \int_0^c g(x)\,dx - g(c) + g(0).$$

Proof: The increasing case is obvious. In the decreasing case, the integral for any unit interval is greater than the value at the right end of the interval: $\int_j^{j+1} g(x)\,dx > g(j+1)$. We obtain the lemma by adding $g(j) - g(j+1)$ to both sides and summing over j. □

Lemma 4 *The code length l for all occurrences of a single symbol in a block in ED order is less than $I_L + \Delta_I + \Delta_U$.*

Proof: We show that $l < S_U \le I_U + \Delta_U = I_L + \Delta_I + \Delta_U$. Since S_U represents the code length for the symbol if all occurrences of the symbol come as late as possible in an ED order, we have $l < S_U$. If $p < q - 1/A$, then $-\lg p_E(x)$ is monotone increasing, so by Lemma 3 we have $S_U < I_U$. If $p > q - 1/A$, then $-\lg p_E(x)$ is monotone decreasing, so by Lemma 3 we have $S_U < I_U + \lg p_E(c) - \lg p_E(0) = I_U + \Delta_U$. If $p = q - 1/A$, then $S_U = I_U$. In all three cases, $S_U \le I_U + \Delta_U$. From the definition of Δ_I, $I_U = I_L + \Delta_I$. □

We now relate I_L to the entropy of the beginning and ending probability distributions. We write $I_L(i)$ to differentiate the values of I_L evaluated for different symbols a_i.

Lemma 5 *Let* $L_H = B\left(\frac{1}{1-f}H(R) - \frac{f}{1-f}H(Q)\right)$. *Then* $L_H = \sum_{i=1}^{k} I_L(i)$.

Proof: By appropriate substitutions, we have

$$\begin{aligned}\sum_{i=1}^{k} I_L(i) &= \sum_{i=1}^{k} B\left(\frac{1}{1-f}(-r_i \lg r_i) - \frac{f}{1-f}(-q_i \lg q_i)\right) \\ &\quad + \sum_{i=1}^{k} B\left(\frac{f}{(1-f)^2}(q_i - r_i)\right).\end{aligned}$$

The last sum is 0 because Q and R are probability distributions, so $\sum_{i=1}^{k} q_i = \sum_{i=1}^{k} r_i = 1$. The lemma follows from the definition of $H(\cdot)$. □

Finally we bound the per-block error.

Lemma 6 *Let* L_j *be the compressed length of block* j. *Then*

$$L_j \le B\left(\frac{1}{1-f}H(R) - \frac{f}{1-f}H(Q) + k \lg \frac{B}{km}\right).$$

Proof: From Lemmas 4 and 5, we have $L_j \le L_H + \sum_{i=1}^{k}(\Delta_I + \Delta_U)$. A scaling factor f of $1/2$ implies that $A = B$. Asymptotics under this condition give

$$\Delta_I + \Delta_U = \begin{cases} 1 - O(1/c) & \text{if } p \le q - 1/A; \\ \lg(c/s) + O((s+1)/c) & \text{if } p > q - 1/A. \end{cases}$$

The sum $\sum_{i=1}^{k}(\Delta_I + \Delta_U)$ is maximized when as many symbols as possible have large c and small s; the sum's largest possible value cannot exceed its value when $s = m$ and $c = B/k$ for all k symbols, in which case $\Delta_I + \Delta_U = \lg(B/km) + O((m+1)/c)$. We obtain the result by setting $m = 1$ and summing over the symbols. □

The proof of the upper bound in Theorem 3 follows from Lemma 6 by summing over all blocks. (We are neglecting any special effects of a longer first block or shorter last block.) There is much cancellation because $H(R)$ of one block is $H(Q)$ of the next.

3.1.2 Proof of the lower bound

In the following proof of the lower bound of Theorem 3 we append a prime to the label of each definition and lemma to show the correspondence with the definition and lemmas used in the proof of the upper bound.

Definition 2′ A block of length B containing $c_1, c_2, \ldots, c_k$ occurrences of symbols $a_1, a_2, \ldots, a_k$, respectively, has the *reverse ED* property if for each symbol a_i and for all m, $1 \le m \le c_i$, the symbol occurs for the mth time not before position $(m-1)B/c_i$.

Lemma 2′ *Every distribution of symbol counts has an order with the reverse ED property.*

Proof: By Lemma 2 there is always an order with the ED property. Such an order, when reversed, has the reverse ED property. □

Lemma 3′ *If $g(x)$ is monotone decreasing, then*

$$\sum_{j=0}^{c-1} g(j) > \int_0^c g(x)\,dx.$$

If $g(x)$ is monotone increasing, then

$$\sum_{j=0}^{c-1} g(j) > \int_0^c g(x)\,dx - g(c) + g(0).$$

Proof: Similar to that of Lemma 3. □

Lemma 4′ *The code length l for all occurrences of a single symbol in a block in reverse ED order is greater than $I_L - \Delta_L$.*

Proof: We show that $l > S_L \ge I_L - \Delta_L$. Since I_L represents the code length for the symbol if all occurrences of the symbol come as early as possible in a reverse ED order, we have $l > S_L$. If $p > q$, then $-\lg p_B(x)$ is monotone decreasing, so by Lemma 3′ we have $S_L > I_L$. If $p < q$, then $-\lg p_B(x)$ is monotone increasing, so by Lemma 3′ we have $S_L > I_L + \lg p_B(c) - \lg p_B(0) = I_L - \Delta_L$. If $p = q$, then $S_L = I_L$. In all three cases, we have $S_L \ge I_L - \Delta_L$. □

Lemma 6′ *Let L_j be the compressed length of block j. Then*

$$L_j \ge B\left(\frac{1}{1-f}H(R) - \frac{f}{1-f}H(Q)\right) - k.$$

Proof: From Lemmas 4′ and 5, $L_j \ge L_H - \sum_{i=1}^{k} \Delta_L(i)$. Asymptotics when $f = 1/2$ give $\Delta_L = 1 - O(c/s)$, which has maximum value 1, so the sum is at most k. □

The proof of the lower bound in Theorem 3 follows from Lemma 6′ by summing over all blocks, noting that $b = t/B$ and $m = 1$.

3.1.3 Non-scaling corollary

By letting $B = t$, $f = n/(t+n)$, and $m = 1$ in Lemmas 6 and 6′, we obtain the code length without scaling:

Corollary 1 *When we do not scale at all, the code length L_{NS} satisfies:*

$$tH_{\text{final}} + n(H_{\text{final}} - H_0) - k < L_{NS} < tH_{\text{final}} + n(H_{\text{final}} - H_0) + k\lg(t/k).$$

We can get important insights by contrasting upper bounds in this corollary and Theorem 3. Scaling will bring about a shorter encoding by tracking the block-by-block entropies rather than matching a single entropy for the entire file, but when we forgo scaling the overhead is less, proportional to $\lg t$ instead of to t. Scaling will do worst on a homogeneous file, but even then the overhead will increase the code length by only about $(k/B)\lg(B/k)$ bits per input symbol, less than 0.1 bit per symbol for a typical file.

4 Application to Higher Order Models

We now extend our results to higher order models. Witten and Cleary [21] present an adaptive method called *prediction by partial matching* (or *PPM*) in which they maintain models of various orders. At each point they use the highest-order model in which the symbol has occurred in the current context, with a special *escape* symbol to indicate the need to drop to a lower order. (Because most contexts occur only a few times with only a few different symbols, assigning an initial weight of 1 to each alphabet symbol as we did in Section 2 is an unsatisfactory solution to the zero-frequency problem in higher-order models. Doing so would give too much weight to symbols that never occur.) See [7] or [22] for a detailed description of the PPM method. Witten, Cleary, Moffat, and Bell have proposed at least five methods for estimating the probability of the escape symbol [14,21,23]. All of the methods give approximately the same compression; PPMB [21] is the most readily analyzed.

In PPMB, in each context the escape event is treated as a separate symbol with its own weight and probability; the first occurrence of an ordinary symbol is not counted and the first two occurrences are coded as escapes. Treating the escape event as a normal symbol, we can apply the results of Section 3 if we make adjustments for the first two occurrences of each symbol, since in PPMB the code length is independent of the order of the symbols.

In the block in which a given symbol occurs for the first time, we can take the occurrences to be evenly distributed in the sense of Definition 2, with symbol weights (numerators) running from 1 to c and occurrence positions (denominators) running from $A+B/c$ to $A+B$. If this were coded in the normal way, the code length would be bounded above by Lemmas 4 and 5. Since the mechanism of PPMB excludes the last two numerators, $c-1$ and c, and the first two denominators, $A+B/c$ and $A+2B/c$, the code length from the approximation must be adjusted by adding $L_{\textbf{adjustment}}$:

$$\begin{aligned}
L_{\textbf{adjustment}} &= L_{\textbf{actual}} - L_{\textbf{estimated}} \\
&= \lg \frac{(c-1)c}{(A+B/c)(A+2B/c)} \\
&= 2\lg c - 2\lg A + \lg\left(1-\frac{1}{c}\right) - \lg\left(1+\frac{1/f-1}{c}\right) \\
&\quad -\lg\left(1+2\frac{1/f-1}{c}\right).
\end{aligned}$$

We must also adjust the entropy: the actual value of the initial probability q is 0 instead of $1/A$, and the actual value of the final probability r is $(c-1)/(A+B)$ instead of $(c+1)/(A+B)$. For convenience we denote the entropy term $B\left(\frac{1}{1-f}H(R) - \frac{f}{1-f}H(Q)\right)$ by h. We define and compute the adjustment:

$$\begin{aligned}
h_{\textbf{adjustment}} &= h_{\textbf{actual}} - h_{\textbf{estimated}} \\
&= c\lg\left(1+\frac{2}{c-1}\right) + 2\lg c - \lg A + 2\lg f + \lg\left(1-\frac{1}{c^2}\right).
\end{aligned}$$

The code length is then given by

$$\begin{aligned}
L_{\textbf{actual}} &= h_{\textbf{actual}} + (L_{\textbf{adjustment}} - h_{\textbf{adjustment}}) + \text{small terms} \\
&= h_{\textbf{actual}} + c\lg\left(1+\frac{2}{c-1}\right) + \lg A + 2\lg f + \lg\left(1+\frac{1}{c}\right) \\
&\quad + \lg\left(1+\frac{1/f-1}{c}\right) + \lg\left(1+2\frac{1/f-1}{c}\right) \\
&\quad + \text{small terms}.
\end{aligned}$$

The net adjustment is negative when $c=3$, the smallest possible value of c, and increases with c. The largest value occurs when $c \gg A$, that is, when the number of occurrences of a new symbol exceeds the initial weight of all the symbols; in this unlikely event the net adjustment is positive but less than $2\lg c - 3\lg A$.

Now we can extend Theorem 3 to the PPMB model with scaling:

Theorem 4 *When we use PPMB with scaling, the code length L is bounded by*

$$L < \sum_{\text{contexts } X} \left(B\left(\sum_{m=1}^{b_X} H_{X,m} + H_{X,b} - H_{X,0}\right) + O(k_X t_X)\right).$$

Proof: The proof follows from the discussion above, noting that the maximum adjustment for one symbol a_i in one context X is $O(\lg c_{X,i})$ and that $\lg c_{X,i} < c_{X,i} < t_X$. □

5 Coding Effects

In this section we prove that coding effects (as distinguished from modeling effects) are insignificant, and hence that our assumption of exact coding is appropriate. We improve on earlier treatments of the coding effects [3,7], which appeal to experimental evidence rather than proof.

5.1 Rounding counts to integers

In Section 3 we analyzed the modeling effect of periodic scaling; here we analyze the coding effect. Witten, Neal, and Cleary scale the counts to avoid register overflow, and to prevent any count from falling to 0, they round fractional counts to the next higher integer. This gives more weight to symbols whose count happens to be odd when scaling occurs.

Theorem 5 *Rounding counts up to the next higher integer increases the code length for the file by no more than $n/2B$ bits per input symbol.*

Proof: Each symbol whose weight is rounded up causes the denominators of all probabilities in the next block to be too large, by $1/2$. If r is the number of symbols subject to roundup, $r/2$ of the denominators in computing the coding interval will be approximately T instead of $T/2$, each adding one bit to the code length of the block, so the block's code length will be $r/2$ bits longer. The effect for the entire file (t/B blocks) is $rt/2B$ bits, or, since $r \leq n$, at most $n/2B$ bits per input symbol. □

This effect is typically 0.02 bit or less per input symbol.

5.2 Using integer arithmetic

Witten, Neal, and Cleary use integers from a large fixed range, typically $[0, 8B]$, instead of using exact rational arithmetic, and they transmit encoded bits as soon as they know them instead of waiting until the entire string has been encoded, scaling up the range to represent only that half of the original range whose identity has not yet been transmitted. The result is nearly the same compression efficiency as with exact rational arithmetic.

As is apparent from the following description of the coding section of the Witten–Neal–Cleary algorithm, scaling up the range is only approximate:

1. We select a subrange of the current interval $[low, high]$ whose length within $[low, high]$ is proportional to p.

2. We repeat the following steps as many times as possible:

    (a) If the new subrange is not entirely in the lower, middle, or upper half of the full range of values, we return.

(b) If the new subrange is in the lower or upper half, we output 0 or 1 respectively, plus any bits left over from previous symbols. If the subrange is entirely in the middle half, we keep track of this fact for future output.

(c) We scale the subrange up:

i. We shift the subrange to ignore the part of the full range in which the subrange is known not to lie.

ii. We double both *low* and *high* and add 1 to *high*.

In this algorithm, any roundoff error in selecting the first subrange will propagate through the entire scaling up process. In the worst case, a symbol with a count of 1 could result in a subrange of length 1, even though the unrounded subrange size might be just below 2. In effect, this would assign a symbol probability of only half the correct value, resulting in a code length one bit too long. In practice, however, the code length error in one symbol is seldom anywhere near that large, and because the errors can be of either sign and have an approximately symmetrical distribution with mean 0, the average error is usually very small. Witten, Neal, and Cleary empirically estimate it at 10^{-4} bit per input symbol.

In order to get a rigorous bound on the compression loss, we analyze a new algorithm that maintains full precision when scaling up the range. Instead of adding 1 to *high* at each step, we add either 0 or 1 to *low* and independently to *high* according to the exact result of the initial subrange selection. To prevent the overlap of subranges from different alphabet symbols, we add 1 to *low* and subtract 1 from *high* when no more scaling is possible. This last step always makes the code length longer than that of exact arithmetic coding, but by a tiny amount, as shown in the following theorem:

Theorem 6 *When we use the high precision algorithm for scaling up the subrange, the code length is greater than the ideal code length $-\lg p$ by an amount bounded by $2/(B \ln 2)$ bits per input symbol.*

Proof: After the scaling up process, the smallest possible subinterval size is $2B$. As a result of rounding while scaling up, *low* could be too high by as much as 1, and *high* could be too low by as much as 1, so the subinterval could be too short by as much as 2. (It could also be too long by as much as 2.) The final overlap-prevention step always shortens the subinterval by 2, so the subinterval is always too short, possibly by as much as 4. An interval too short by 4 could be too short by a factor of as much as $(2B-4)/2B$, which corresponds to a code length increase of $-\lg(B-2)/B \sim 2/(B \ln 2)$. □

A loss of $2/(B \ln 2)$ bits per input symbol is negligible, about 4×10^{-4} bit per symbol for typical parameters. In practice the high precision algorithm gives exactly the same compression as the algorithm of Witten, Neal, and Cleary, but its compression loss is *provably* small.

5.3 Encoding end-of-file

When using arithmetic coding, we must explicitly signal the end of the file. Witten, Neal, and Cleary introduce a special low-weight symbol in addition to the normal alphabet symbols; it is encoded once, at the end of the file. In the following theorem we bound the cost of identifying end-of-file by this method:

Theorem 7 *The use of a special end-of-file symbol results in additional code length of less than $t/(B \ln 2) + \lg B + 10$ bits.*

Proof: The cost has four components:

- at most $\lg B + 1$ bits to encode the end-of-file symbol (since its probability must be at least as large as the smallest possible probability, $1/2B$)
- fewer than $t/(B \ln 2)$ bits in wasted code space to allow end-of-file at any point (each probability can be reduced by a factor of between $(2B - 1)/2B$ and $(B - 1)/B$, resulting in a loss of between $-\lg((2B - 1)/2B) \approx 1/(2B \ln 2)$ and $-\lg((B - 1)/B) \approx 1/(B \ln 2)$ bits per symbol)
- two disambiguating bits after the end-of-file symbol
- up to seven bits to fill the last byte.

$\square$

An alternative, transmitting the length of the original file before its encoding, reduces the cost to between $\lg t$ and $2 \lg t$ bits by using an appropriate encoding of integers [11,12,13], but requires the file length to be known before encoding can begin.

The end-of-file cost using either of these methods is negligible for a typical file, less than 0.01 bit per input symbol.

6 Conclusion

We have proven that adaptive codes are as good as semi-adaptive codes, in particular the decrementing semi-adaptive codes. Using our notion of weighted entropy, we have exactly characterized the tradeoff between the overhead associated with scaling and the savings that it can realize by exploiting locality of reference. The largest code length savings come from more sophisticated (higher order) models such as PPMB, and our scaling analysis extends accordingly. We have proven that computational effects in arithmetic coding are small, so we can treat practical arithmetic coders as though they were exact coders.

References

[1] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the Institute of Radio Engineers*, 40, pp. 1098–1101, 1952.

[2] G. G. Langdon, "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.*, 28, no. 2, pp. 135–149, Mar. 1984.

[3] I. H. Witten, R. M. Neal and J. G. Cleary, "Arithmetic Coding for Data Compression," *Comm. ACM*, 30, no. 6, pp. 520–540, June 1987.

[4] J. J. Rissanen, "Generalized Kraft Inequality and Arithmetic Coding," *IBM J. Res. Develop.*, 20, no. 3, pp. 198–203, May 1976.

[5] R. Pasco, "Source Coding Algorithms for Fast Data Compression," Stanford Univ., Ph.D. Thesis, 1976.

[6] F. Rubin, "Arithmetic Stream Coding Using Fixed Precision Registers," *IEEE Trans. Inform. Theory*, IT-25, no. 6, pp. 672–675, Nov. 1979.

[7] T. C. Bell, J. G. Cleary and I. H. Witten, *Text Compression*. Englewood Cliffs, NJ, Prentice-Hall, 1990.

[8] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.*, 27, pp. 398–403, July 1948.

[9] J. G. Cleary and I. H. Witten, "A Comparison of Enumerative and Adaptive Codes," *IEEE Trans. Inform. Theory*, IT-30, no. 2, pp. 306–315, Mar. 1984.

[10] A. S. Fraenkel and S. T. Klein, "Robust Universal Complete Codes as Alternatives to Huffman Codes," Dept. of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, Technical Report, 1985.

[11] R. G. Stone, "On Encoding of Commas Between Strings," *Comm. ACM*, 22, no. 5, pp. 310–311, May 1979.

[12] P. Elias, "Universal Codeword Sets and Representations of Integers," *IEEE Trans. Inform. Theory*, IT-21, no. 2, pp. 194–203, Mar. 1975.

[13] M. Wang, "Almost Asymptotically Optimal Flag Encoding of the Integers," *IEEE Trans. Inform. Theory*, IT-34, no. 2, pp. 324–326, Mar. 1988.

[14] I. H. Witten and T. C. Bell, "The Zero Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression," Preprint, 1989.

[15] D. E. Knuth, "Dynamic Huffman Coding," *J. Algorithms*, 6, no. 2, pp. 163–180, June 1985.

[16] B. Y. Ryabko, "Data Compression by Means of a Book Stack," *Problemy Peredachi Informatsii*, 16, no. 4, 1980.

[17] J. L. Bentley, D. D. Sleator, R. E. Tarjan and V. K. Wei, "A Locally Adaptive Data Compression Scheme," *Comm. ACM*, 29, no. 4, pp. 320–330, Apr. 1986.

[18] P. Elias, "Interval and Recency Rank Source Coding: Two On-line Adaptive Variable Length Schemes," *IEEE Trans. Inform. Theory*, IT-33, no. 1, pp. 3–10, Jan. 1987.

[19] G. V. Cormack and R. N. Horspool, "Algorithms for Adaptive Huffman Codes," *Inform. Process. Lett.*, 18, no. 3, pp. 159–165, Mar. 1984.

[20] K. Mohiuddin, J. J. Rissanen and M. Wax, "Adaptive Model for Nonstationary Sources," *IBM Technical Disclosure Bulletin*, 28, no. 11, pp. 4798–4800, Apr. 1986.

[21] J. G. Cleary and I. H. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Trans. Comm.*, COM-32, no. 4, pp. 396–402, Apr. 1984.

[22] T. C. Bell, I. H. Witten and J. G. Cleary, "Modeling for Text Compression," *Comput. Surveys*, 21, no. 4, pp. 557–591, Dec. 1989.

[23] A. Moffat, "A Note on the PPM Data Compression Algorithm," Dept. of Computer Science, Univ. of Melbourne, Victoria, Australia, Research Report 88/7, 1988.