LOSSLESS DATA COMPRESSION: AN OVERVIEW

by

SWAPANDEEP SANDHU

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Canada
October 2021

Abstract

The constant growth of data in the digital world leads to the requirement of efficient methods to store and transmit data. Due to limited resources, data compression techniques are used to minimize the size of data being stored or communicated.

This work is devoted to the study and comparison of some lossless data compression methods.

First, we focus on two classical methods which are the Huffman and arithmetic coding methods. These two methods are discussed in detail including their basic properties in the context of information theory. We also highlight their shortcomings as they are not necessarily providing optimal compression nor defining the most efficient framework for data transmission.

We investigate variants of these methods by introducing adaptive Huffman and adaptive arithmetic methods.

A framework for evaluation and comparison of the methods is constructed and applied to the algorithms presented.

Acknowledgements

I would like to thank the Computer Science department at Bishop's University for giving me the opportunity to pursue a Master's degree. I would like to underline the support, patience and guidance received from Dr. Madjid Allili, without him none of this would have been possible.

Thank you to my family, for their enduring love and support, and believing in me when I did not believe in myself

Introduction

Data compression has become an indispensable part of our lives. It allows us to represent data in compact form and therefore store and transmit more data with the same cost. We generate and use an ever-increasing amount of data in digital form. Technological advancements in digital communication have allowed us to transmit data to almost anyone on the globe. Overall, the need for mass storage and transmission seems to increase twice as fast as the storage and transmission capacities improve [1]. The fact that information is increasingly represented in a digital form and its transmission happens mainly electronically leads to the requirement of efficient techniques to represent, store, and transmit data. There are many ways of representing information digitally, and any particular choice of representation is called a code. The object of data compression is to design representations that minimize the amount of data needed to represent a given information.

In the last ten years, we have seen a number of papers covering new approaches to data compression. For instance, Minnen et al. [2] implement lossy compression using variational auto-encoder model, and Tschannen et al. [3] train a model to achieve an optimal lossy compression method. New trends in lossless compression have been less covered and much of recent literature on lossless data compression has been in form of survey papers [4, 5]. However, some very recent work by Vanderberg et al. [6] and Ruan et al. [7] can be listed among the efforts to improve classical methods of lossless compression.

In this thesis, we focus on compression without loss of information, known as lossless compression. We study two classical methods of compression which are the Huffman and arithmetic coding methods [1, 8]. These two methods are discussed in detail and their shortcomings are highlighted. These two methods, called static methods, assign codewords to source messages based on the probabilities with which the source messages appear in the message ensemble. They both require two passes, where the first pass is used to read all the message and build the probability model of the source, and the second pass is to do the encoding. In some instances this may be an immense amount of data or the data may not be available entirely. There may be a need to start the encoding before the whole data is read. Adaptive coding schemes which were created to deal with this type of problem are also discussed. The probability models are estimated and updated as more input is processed. The designed code is changing so as to remain optimal for the current estimates.

In Chapter 1, we introduce the main concepts of information theory and coding. The definitions and concepts necessary to the study and evaluation of data compression methods are discussed.

In Chapter 2, we discuss different source and data models. We introduce the main classes of sources which are the memoryless sources and sources with memory. We explain that the Markov chains can be exploited to take advantage of interaction between groups of symbols and provide models to the sources with memory.

In Chapter 3, we develop the details of the coding strategies for the static Huffman and arithmetic methods and relate them to the basic concepts discussed in Chapter 1. We provide a concise algorithm with the complexity analysis of the Huffman coding method. We discuss the main properties

of the two methods and illustrate them with simple examples.

Chapter 4 is devoted to the discussion of adaptive Huffman and arithmetic coding methods. We describe the procedures that convert these algorithms into one-pass methods. We highlight the essential modifications of the algorithms so that the encoder learns the probability distributions based on the statistics of the symbols already encountered as more input is processed and the coding progresses.

In Chapter 5, we carry out experiments on these algorithms using various methods aimed at validating and comparing their performances. In addition to validating and confirming the projected performances for the individual methods, we find out that the adaptive arithmetic coding provides the best compression out of these four tested schemes.

Contents

Chapter 1

Fundamental Concepts of Data Compression

In this chapter, we discuss the basic notions necessary to the development and evaluation of data compression methods.

1.1 Information Theory Concepts

Generally speaking, we can define information as facts provided or learned about something or someone. In the context of computing and communication, we can see information as the knowledge conveyed or represented by a particular arrangement or sequence of symbols.

Information Theory is a branch of mathematics that was introduced in the late 1940s with the work of Claude Shannon at Bell Labs [9]. The main objective of this theory is to answer various questions about information, including different ways of storing and communicating messages.

In general, data compression consists of taking a sequence of symbols and transforming them into codes. If the compression is effective, the resulting sequence of codes will be smaller than the original one. Data compression is closely related to the field of Information Theory because of its concern with redundancy. Redundant information in a message takes extra symbols to encode, and if we can get rid of that extra information, we will have reduced the size of the message.

In this section we introduce the main tools to measure the amount of information contained in an information source and any message or sequence of symbols taken from the source. Our discussion is based on the following references [1, 9, 10]

1.1.1 Source of Information

We can view a source of information as an ordered pair $\mathcal{S} = (S, P)$, where $S = \{s_1, s_2, \ldots, s_n\}$ is a finite set of symbols, also known as the source alphabet, and P is a probability distribution defined on the subsets of S. The probability of a symbol s_i is denoted by p_i or $P(s_i)$ and is supposed to reflect the frequency of occurrence of s_i. The probability of a subset A of S is the sum all the probabilities p_i where $s_i \in A$.

A source $\mathcal{S} = (S, P)$ is sampled by choosing an element at random according to the probability distribution P. The probability that the element s_i is chosen is p_i. Note that before the sampling,

there is some uncertainty associated with the outcome, and after the sampling, a certain amount of information is gained about the source.

The quantity of information in a source is given by its entropy.

1.1.2 Entropy

Information Theory uses the term entropy as a measure of how much information is encoded in a message. Shannon borrowed the definition of entropy from thermodynamics and statistical physics (see [10]), where entropy represents the randomness or disorder of a system. A system is assumed to have a set of possible states it can be in, and at a given time there is a probability distribution over those states. In thermodynamics, the entropy of a system is defined as

$$H(S) = \sum_{s \in S} p(s) \log_2 \left(\frac{1}{p(s)} \right), \tag{1.1}$$

where S is the set of possible states, and $p(s)$ is the probability of state $s \in S$. In the context of information theory Shannon simply replaced "state" with "message".

Given a source $\mathcal{S} = (S, P)$, where P is a discrete probability distribution with weights $p_1, \ldots, p_n$ satisfying

$$0 \leq p_i \leq 1, \text{ for } 1 \leq i \leq n, \text{ and } \sum_{i=1}^{n} p_i = 1,$$

we would like to define a function $H(p_1, p_2, \ldots, p_n)$ that measures the uncertainty involved in sampling from $\mathcal{S}$. H will be called the entropy of $\mathcal{S}$ or the entropy of the probability distribution P. This stems from the fact that in Equation 1.1, the formula depends only on the probability distribution and not on the states themselves.

We can use Equation 1.1 directly to define H, however it will be beneficial to develop some of the reasoning and the mathematical steps that lead to the definition of $H(p_1, p_2, \ldots, p_n)$ in our context. First, we want H to depend only on the probabilities $p_1, \ldots, p_n$ and not on the elements of the source alphabet. Moreover, we want H to be continuous in these variables so that a small change in the probabilities will produce only a small change in uncertainty or entropy. In addition, when all outcomes are equally likely, it seems reasonable to ask that the more outcomes there are, the greater should be the entropy, that is

$$H\left(\frac{1}{n}, \ldots, \frac{1}{n}\right) < H\left(\frac{1}{n+1}, \ldots, \frac{1}{n+1}\right).$$

It is important to note here that H does not depend on a fixed number of variables as it is a generic function having one variable P representing the probability distribution.

There is one additional property for H. Assume that the source symbols $S = \{s_1, \ldots, s_n\}$ are partitioned into nonempty subsets $E_1, E_2, \ldots, E_m$, where $|E_i| = e_i$ with $\sum_{i=1}^{m} e_i = n$. Consider the following experiment. We first choose a subset E_i with a probability proportional to its size, i.e. $P(E_i) = \frac{e_i}{n}$, and then we pick an element with equal probability from E_i. If s_j is in the subset E_k then the conditional probability of occurrence of s_j given E_i is given by

$$P(s_j|E_i) = \begin{cases} 0 & \text{if } i \neq k \\ \frac{1}{e_k} & \text{if } i = k \end{cases}$$

and using the total probability formula, we have

$$P(s_j) = \sum_{i=1}^{n} P(s_j|E_i)P(E_i) = \left(\frac{1}{e_k}\right)\left(\frac{e_k}{n}\right) = \frac{1}{n},$$

which implies that the probability of choosing s_j in this experiment os the same as if we choose directly from S with equal probability. Therefore, the uncertainty in the outcomes should also be the same.

The uncertainty when choosing directly from S with equal probability is $H\left(\frac{1}{n},\ldots,\frac{1}{n}\right)$. In the new experiment, the uncertainty of choosing one subset is $H\left(\frac{e_1}{n},\ldots,\frac{e_m}{n}\right)$ and, once a subset is chosen, we still have the uncertainty involved in choosing an element from that subset. The expected (average) uncertainty in this process is

$$\sum_{i=1}^{m} P(E_i) \times \text{ (uncertainty in choosing from } E_i) = \sum_{i=1}^{m} P(E_i) H\left(\frac{1}{e_i},\ldots,\frac{1}{e_i}\right).$$

Thus, we get the property

$$H\left(\frac{1}{n},\ldots,\frac{1}{n}\right) = H\left(\frac{e_1}{n},\ldots,\frac{e_m}{n}\right) + \sum_{i=1}^{m} \frac{e_i}{n} H\left(\frac{1}{e_i},\ldots,\frac{1}{e_i}\right).$$

In summary, we want the function H to have the following properties.

(1) $H(p_1, p_2, \ldots, p_n)$ is defined and continuous for all $p_1, \ldots, p_n$ satisfying

$$0 \le p_i \le 1, \text{ for } 1 \le i \le n, \text{ and } \sum_{i=1}^{n} p_i = 1.$$

(2) We have

$$H\left(\frac{1}{n},\ldots,\frac{1}{n}\right) < H\left(\frac{1}{n+1},\ldots,\frac{1}{n+1}\right), \quad \text{for } n \in \mathbb{N}.$$

(3) For $e_i \in \mathbb{N}$, $\displaystyle\sum_{i=1}^{m} e_i = n$

$$H\left(\frac{1}{n},\ldots,\frac{1}{n}\right) = H\left(\frac{e_1}{n},\ldots,\frac{e_m}{n}\right) + \sum_{i=1}^{m} \frac{e_i}{n} H\left(\frac{1}{e_i},\ldots,\frac{1}{e_i}\right).$$

It turns out that the properties above define uniquely a function H as stated in the following proposition.

Proposition 1.1. *A function H satisfies properties (1), (2), and (3) if and only if it has the form*

$$H_b(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_b p_i \tag{1.2}$$

where $b > 1$, and where we set $p\log_b p = 0$ for $p = 0$.

Let $P = \{p_1, \ldots, p_n\}$ be a probability distribution. Then the quantity

$$H_b(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_b p_i = \sum_{i=1}^{n} p_i \log_b \frac{1}{p_i}$$

is called the b-ary entropy of the distribution P. If $\mathcal{S} = (S, P)$ is a source with $P(s_i) = p_i$, then we refer to $H_b(\mathcal{S}) = H_b(p_1, \ldots, p_n)$ as the entropy of $\mathcal{S}$.

In many applications and many important books of information theory, base $b = 2$ is chosen. We talk then about binary entropy and use the notation $H(p_1, \ldots, p_n)$ for binary entropy.

Concerning the units of entropy, observe that if we set $S = \{0, 1, 2, \ldots, n-1\}$, then it is reasonable to assume that sampling from S with equal probability gives an amount of information equal to one n-ary unit. For instance if $S = \{0, 1\}$, then sampling from S with equal probability gives one binary unit (one *bit*) of information. Since

$$H_n\left(\frac{1}{n}, \ldots, \frac{1}{k}\right) = -\sum_{i=n}^{k} \frac{1}{n} \log_n \frac{1}{n} = 1,$$

we see that H_n measures the number of n-ary units of information. Thus, the binary entropy H_2 measures information in *bits* and H_e, where $e = 2.718\ldots$ is the base of the exponential, measures information in natural units, or *nats*.

For example, if we sample from the set $S = \{s_1, s_2, s_3, s_4\}$ with equal probabilities $p_i = \frac{1}{4}$ yields

$$H_2(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}) = \sum_{i=1}^{4} \frac{1}{4} \log_2 4 = \log_2 4 = 2 \quad \text{bits.}$$

Sampling from the set $S = \{s_1, s_2, s_3, s_4\}$ with probabilities $p_1 = p_2 = p_3 = \frac{1}{8}$ and $p_4 = \frac{5}{8}$ yields

$$H_2(\frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{5}{8}) = \frac{1}{8}\log_2 8 + \frac{1}{8}\log_2 8 + \frac{1}{8}\log_2 8 + \frac{5}{8}\log_2 \frac{8}{5} = \frac{9}{8} + \frac{5}{8}(0.678) = 1.548 \quad \text{bits.}$$

Since we are a little bit more certain about the outcome in the second case, the entropy is smaller.

Shannon [9] also defined the notion of *self information* of a source $\mathcal{S} = (S, P)$ as the entropy of the source, and more generally the *self information* of a message s as the quantity

$$i(s) = \log_2 \frac{1}{P(s)},$$

where $p(s)$ is the probability of the message s. This self information represents the number of bits of information contained in s and, roughly speaking, the number of bits that should be used to encode s. This also indicates that messages with higher probability will contain less information. It follows that the entropy is simply a probability weighted average of the self information of each message from a given source.

In addition to the mathematical derivation of the logarithm formula for the entropy from specified properties, we can also see intuitively that the inverse probability is the right measure for self information of a message and this is also related to to message length. For instance, if a set contains $n = 2^k$ messages of equal probability, the probability of each is $\frac{1}{n}$. Moreover, if all the messages are of the same length, then $\log_2 n = k$ bits are required to represent each message. This is exactly the self information since

$$i(s_k) = \log_2 \frac{1}{p_k} = \log_2 \frac{1}{\frac{1}{n}} = \log_2 n = \log_2 2^k = k.$$

For example, if there are $8 = 2^3$ messages $s_1, \ldots, s_8$, then we can use the following binary representations with length 3: $000, 001, 010, 011, 100, 101, 110, 111$ for the messages.

Another property of the mathematical definition of information that is intuitively relevant is that the information of the occurrence of two independent messages must be the sum of the information given by each. In particular if messages A and B are independent, the probability of sending one after the other is $P(A \cap B) = P(A)P(B)$ and the information contained in them is

$$\begin{aligned} i(A \cap B) &= \log_2 \frac{1}{P(A \cap B)} \\ &= \log_2 \frac{1}{P(A)P(B)} \\ &= \log_2 \frac{1}{P(A)} + \log_2 \frac{1}{P(B)} \\ &= i(A) + i(B) \end{aligned}$$

The logarithm is the most natural and simplest function that satisfies this property.

1.2 Coding

Data compression is often defined as coding (or encoding) of data, where coding is a general term referring to any special representation of data that satisfies a given need. We will give here the basic definitions and terminology (see [11, 12]).

Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a finite set, which we refer to as an *alphabet*. A *word*, or string, over the alphabet $\mathcal{A}$ is any sequence of elements of $\mathcal{A}$.

We will write words of the alphabet in the form

$$\mathbf{w} = a_{i_1} \ldots a_{i_k}$$

as a concatenation of symbols of the alphabet $\mathcal{A}$. Other notations are possible where for readability reasons spaces, commas, parentheses or other punctuation marks between the symbols are introduced.

We define the empty word θ as the unique word with no symbols. The length of a word $\mathbf{w}$, denoted $\texttt{len}(\mathbf{w})$ or $\ell(w)$, is by definition the number of alphabet symbols in the word. The set of all words over $\mathcal{A}$ is denoted by $\mathcal{A}^*$.

Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a finite set called the *code alphabet*. An n-ary *code* (often called just a *code*) is a non empty subset C of the set $\mathcal{A}^*$ of all words over $\mathcal{A}$. The size n of the code alphabet is called the *radix* of the code, and the elements, words, or strings of the code are called *codewords*. A code with alphabet is $\{0, 1\}$ is called a *binary code*.

Let $\mathcal{S} = (S, P)$ be a source. An *encoding scheme* for $\mathcal{S}$ is an ordered pair (C, f), where C is a code and $f : S \to C$ is an injective function, called an *encoding function*. An encoding function is therefore a function that assigns a unique codeword from C to each source symbol in S.

1.2.1 Average Codeword Length

One of the measures of efficiency of an encoding scheme is the *average codeword length* defined as follows. The average codeword length of an encoding scheme (C, f) for a source $\mathcal{S} = (S, P)$,

where $S = \{s_1, \ldots, s_n\}$, is defined by

$$AverageLength(C, f) = \sum_{i=1}^{n} P(s_i)\texttt{len}(f(s_i))$$

Consider the source $\mathcal{S} = (S, P)$, where $S = \{s_1, s_2, s_3\}$ and probabilities $p_1 = \frac{3}{15}$, $p_2 = \frac{3}{15}$, and $p_3 = \frac{9}{15}$. Assume we are using two encoding schemes (C_1, f_1) and (C_2, f_2), where

$$\begin{array}{ll} C_1 = \{00, 100, 111\}, & C_1 = \{10, 11, 10001\} \\ f_1(s_1) = 100, & f_2(s_1) = 10001 \\ f_1(s_2) = 00, & f_2(s_2) = 11 \\ f_1(s_3) = 111, & f_2(s_3) = 10 \end{array}$$

Since

$$AverageLength(C_1, f_1) = \frac{3}{15}(3) + \frac{3}{15}(2) + \frac{9}{15}(3) = \frac{42}{15},$$

and

$$AverageLength(C_2, f_2) = \frac{3}{15}(5) + \frac{3}{15}(2) + \frac{9}{15}(2) = \frac{39}{15}.$$

It follows that (C_2, f_2) has a smaller average codeword length, and so is more efficient than (C_1, f_1), even though the code C_2 has longer codewords on the average than code C_1. This example shows that the average codeword length of a code is not the same quantity as the average codeword length of an encoding scheme which is a weighted average that depends on the probability distribution P of the source alphabet. Note that it makes sense to compare the average codeword lengths of different encoding schemes only when the corresponding codes have the same radix. For in general, the larger the radix, the shorter we can make the average codeword length.

1.2.2 The Coding Problem and Variable Length Coding

Assume for example we are given M symbols $s_1, \ldots, s_M$ and we want to represent them using bit strings as codewords. The coding problem is obviously to assign unique codewords (otherwise, the representation would not be very useful since we would not be able to recover the original symbols) for each of the symbols using as few bits per symbol as possible. In the absence of any additional information and if we assume that the symbols have equal probability of occurrence, then we need about $\log_2 M$ bits to represent each message. For example, if there are $256 = 2^8$ symbols (think of grey levels), then we need 8 bits. On the other hand, if there are 26 symbols (think of the letters of the alphabet), then since $\log_2 26 \approx 4.7$, we need 5 bits per symbol to represent the letters as with 4 bits we can only represent 16 symbols. If the parameters of the representation are as fixed above, we cannot in general do any better than $\log_2 M$ without additional assumptions. However, if we know something more about the symbols, then we can do better in general. The simplest example is if some symbols are more likely to occur than others, although all of them are assumed to be random and independent of each other.

We will explore here the assignment of variable length codes for efficient coding. More generally, the codes will be categorized as block-to-block, block-to-variable, variable-to-block, or variable-to-variable, where block-to-block indicates that the source messages and codewords are of fixed length and variable-to-variable codes map variable-length source messages into variable-length codewords.

A code C in which all the codewords have the same length is called a *fixed length code*, or *block code*. If a code C contains codewords of different lengths, it is called a *variable length code*. When the probability distribution P is not uniform, variable length coding is usually more efficient than fixed length coding, as illustrated in the following example.

Consider a source $\mathcal{S} = (S, P)$, where $S = \{s_1, s_2, s_3\}$ and the probability distribution satisfies $P(s_1) = 1 - \varepsilon$ and $P(\{s_2, s_3\}) = \varepsilon > 0$. A fixed length binary code must have codeword length of at least 2 in order to encode 3 words. Its average codeword length is

$$\sum_{i=1}^{3} P(s_i) \times 2 = 2.$$

Using a variable length code, we may for instance assign 0 to s_1 and 10 to s_2, and 11 to s_3. This gives an average codeword length of

$$1 \times (1 - \varepsilon) + 2 \times \varepsilon = 1 + \varepsilon,$$

which is less than 2 if $\varepsilon < 1$.

Variable length encoding schemes can be more efficient than fixed length schemes. However, there may be a potential problem in the decoding as illustrated in the following example.

For the sake of terminology, we recall that the algorithm that constructs the mapping and uses it to transform the source messages into strings of codewords is called the *encoder*. The *decoder* performs the inverse operation, restoring the coded message to its original form.

Example 1.2. Consider a source $\mathcal{S} = (S, P)$, where $S = \{s_1, s_2, s_3\}$, and let (C, f) be a variable length code where $C = \{0, 01, 001\}$ and

$$f(s_1) = 0, \quad f(s_2) = 01, \quad \text{and} \quad f(s_3) = 001.$$

The encoding scheme is not uniquely decodable. Indeed, the codeword string 001 can be decoded as $s_1 s_2$ or as s_3. In order to make this encoding scheme uniquely decodable, we may add extra information such as codeword separators which is not a good solution since it increases the overall length of the encoded messages. This motivates the following definition.

A code is *distinct* if each codeword is distinguishable from every other (i.e., the mapping from source messages to codewords is one to one). A distinct code is *uniquely decodable* if every codeword is identifiable when immersed in a sequence of codewords.

Clearly, each of these features is extremely desirable. The *uniquely decodable* property can be defined more formally as follows. A code C is *uniquely decodable* if whenever $c_1, \ldots, c_k$ and $d_1, \ldots, d_j$ are codewords in C and

$$c_1 \ldots c_k = d_1 \ldots d_j$$

then $k = j$ and $c_i = d_i$ for all $i \in \{1, \ldots, k\}$.

The following example shows that a slight change in the codewords can make a code that is not uniquely decodable into a code that is uniquely decodable.

Example 1.3. Consider again the source $\mathcal{S} = (S, P)$, where $S = \{s_1, s_2, s_3\}$, and let (C, f) be a variable length code where $C = \{1, 01, 001\}$ and

$$f(s_1) = 1, \quad f(s_2) = 01, \quad \text{and} \quad f(s_3) = 001.$$

The code in this example differs from the code in Example 1.2 by replacing the codeword 0 with 1. Observe that every codeword in this new code ends with 1 which acts as a codeword separator.

It follows that when reading a codeword string from left to right, we must decode when and only when we encounter a 1. For instance the string 1100101 is easily decoded as $s_1 s_1 s_3 s_2$ and no other decoding is possible.

The goal here is not to explore general methods proving that a given code is uniquely decodable. We want to focus more on codes with properties such as the ability to decode codewords as soon as they are received as opposed to codes for which it is necessary to wait until the entire message has been received before it is possible to start decoding. This type of code is called instantaneously decodable.

1.2.3 Instantaneous Codes and the Prefix Property

A code is said to be *instantaneously decodable* if each codeword in any string of codewords can be decoded as soon as it is received. If a code is instantaneously decodable, then it is also uniquely decodable. However the converse is not necessarily true.

Consider the code $C = \{1, 10, 100, 000\}$. This code is uniquely decodable since we can decode any string of codewords by reading from right to left as 1 always indicates the beginning of a codeword. However, it is clear that C is not instantaneously decodable.

There is actually a simple way to tell whether a code is instantaneously decodable. A code is said to be a *prefix code* or to have the *prefix property* if no codeword is a proper prefix of any other codeword. Formally, this means that whenever $c = c_1 c_2 \ldots c_n$ is a codeword, then $c_1 \ldots c_k$ is not a codeword for all $1 \leq k \leq n-1$. We can easily determine whether a code C has the prefix property by comparing each codeword with all codewords of greater length to see if it is a prefix of any of them.

Consider for example the code $C_1 = \{0, 10, 110\}$. It is easily seen that 1 is not a prefix of 01 and 001 and 01 is not a prefix of 001. Hence C_1 is a prefix code. On the other hand, the code $C_2 = \{0, 01, 001\}$ is not a prefix code since 0 is a prefix of 01 and 001.

We can easily prove that a code C is *instantaneously decodable* if and only if it is a prefix code (see [11]). This shows the importance of the prefix property. Hence, prefix codes are instantaneously decodable; that is, they have the desirable property that the coded message can be parsed into codewords without the need for lookahead. Using the codeword set $C_1 = \{0, 10, 110\}$ which is a prefix code, we can instantaneously decode all codewords in any message. For instance, if the coded message is 001100 then it must be a composition of the codewords 0, 0, 110 and 0.

1.2.4 Redundancy and the Kraft-McMillan's Inequality

Obviously, it is important and desirable to have a class of codes such as the prefix codes which are instantaneously decodable. However, we may wonder if we will lose the possibility to create shorter codes, more precisely the possibility of designing codes with smaller average codeword lengths, if we restrict ourselves to prefix codes. Fortunately, there are classical results establishing that for any non-prefix uniquely decodable code, we can always find a prefix code with the same codeword lengths. The following proposition is a combination of results, proved first by L.G. Kraft in 1949 and then by McMillan in 1956 (see [13] for details), which gives a simple criterion to determine whether or not there is a prefix code with given codeword lengths.

Proposition 1.4.

1. *If $C = \{c_1, \ldots, c_n\}$ is a uniquely decodable code, then its codeword lengths $\ell_1, \ldots, \ell_n$ must*

satisfy the Kraft-McMillan inequality

$$\sum_{k=1}^{n} 2^{-\ell_k} \leq 1. \tag{1.3}$$

2. *If the numbers $\ell_1, \ldots, \ell_n$ satisfy Kraft's inequality, then there exists a prefix code with codeword lengths $\ell_1, \ldots, \ell_n$.*

These results prove that by restricting ourselves to the family of prefix codes, we are not in danger of overlooking non-prefix uniquely decodable codes that have a shorter average length. This is explicitly written in the following corollary.

Corollary 1.5. *The minimum average codeword length, among all uniquely decodable codes for a source $\mathcal{S}$, is equal to the minimum average codeword length among all prefix codes for $\mathcal{S}$.*

We know that the entropy $H(\mathcal{S})$ of a source $\mathcal{S}$ is the amount of information contained in the source. Since a code for $\mathcal{S}$ captures the information in the source, it should be reasonable to think that the average codeword length of such a code must be at least as large as the entropy $H(\mathcal{S})$. In fact, that is what the Noiseless Coding Theorem proved by Shannon [9] asserts and a short proof of this, in the context of prefix codes, is given in Chapter 3 in the section about the Huffman codes. Moreover, the Noiseless Coding Theorem says that by a suitable encoding, the average codeword length can be made as close to the entropy as desired.

Therefore, we define the redundancy of a source coding as the excess of the length of the encoded output over the theoretical minimum which is equal to the entropy of the source. More formally, given a source (S, P) with $S = \{s_1, \ldots, s_n\}$ with probabilities $P(s_i) = p_i$ and an encoding scheme $C = \{c_1, \ldots, c_n\}$ such that c_i is assigned to s_i, we define the redundancy of C as

$$R(C) = \sum_{i=1}^{n} p_i \ell_i - H(\mathcal{S}) = \sum_{i=1}^{n} p_i \ell_i + \sum_{i=1}^{n} p_i \log p_i,$$

where ℓ_i is the length of the codeword representing the source symbol s_i.

Hence, the redundancy measures the difference between the average codeword length of a code and the average information content of an encoded source. A code with a *minimum* average codeword length for a given discrete probability distribution is said to be a *minimum redundancy code*. Since redundancy is defined to be average codeword length minus entropy and entropy is constant for a given probability distribution, minimizing average codeword length minimizes redundancy.

1.3 Data Compression System

A data compression system can be separated into two major components - the encoder and the decoder. Both components are combinations of data compression or decompression techniques and data modeling. The encoder part consists of the input, the modeling and the encoding blocks. There can also be an optional preprocessing stage where input is transformed to a form more suitable for data compression. Likewise, in the decoder part, in addition to the modeling and decoding blocks, there can be an optional postprocessing stage where the output is transformed to a form more suitable for the representation of the final output data.

For each of these stages of encoding and decoding, there exist various approaches and implementations. Since the components are stand alone, it is possible to interchange them, which allows a certain degree of flexibility when designing a compression system as shown in Figure 1.1.

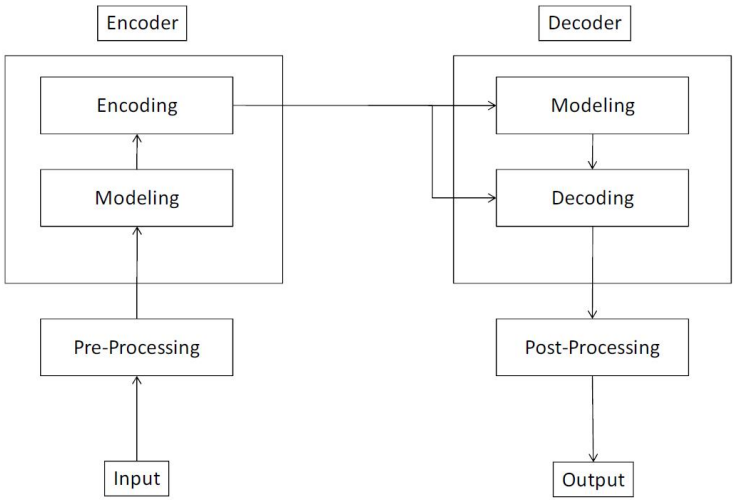

Figure 1.1: Example of a Compression System.

1.3.1 Preprocessing and Postprocessing

Preprocessing transforms and restructures the input in a way that is expected to improve the encoding process. The algorithms in this step are only applicable to specific types of data (text-based, audio, graphics, etc.) and therefore need some sort of input recognition scheme.

Generally, compressed data can be divided into symbolic and diffuse data [14]. Symbolic data is defined as data that can be directly discerned by the human eye. It consists of combinations of symbols, characters and marks. Typical examples of this type of data are text and numeric data. Diffuse data on the other hand cannot be discerned by the human eye. The perceptual interpretation of the data is stored in its structure and cannot be easily extracted. Examples of this type of data include speech, image, and video data.

Specialized techniques exist for all input types. For example, text-based transforms take advantage of characteristics of written languages, like the fact that text consists of sentences and sentences consist of delimited words. Techniques can range from simple substitution of words from a dictionary, to sophisticated procedures such as Word Replacing Transformations [15, 16]. The post-processing reverses the data from the intermediate form to the desirable form for the output.

1.3.2 Modeling

The modeling consists of gathering information about the data that will be used in the encoding or the decoding processes. Modeling the input data is the most important part. It means capturing the structure and redundancy in the data. The better the modeling recognizes the structure of the input, the better it can estimate what its following contents will be. This estimate will in turn affect the performance of the followup encoding and decoding processes.

As a standard example of modeling symbolic data, Prediction by Partial Matching (well known as PPM) is based on the theorems of Shannon [9], which say that the entropy of a source can be better approximated if we track symbol occurrences along with their context. This allows to obtain a good estimation of the real probabilities of symbol occurrence in the input and the entropy coder

can therefore produce a more precise codeword assignment. A more recent method that can be seen as an extension of PPM is Context Mixing [17]. It uses more than one model of estimation. Each one would have its own specifications on how to match the input, and during processing of each bit, each model would give a prediction of the next bit, and a confidence level of this prediction. A mixer would then compute a weighed average and pass the result to an entropy coder. Many other models exist for symbolic and diffuse data.

1.3.3 Coding and Decoding

Entropy coding is the process of assigning bit strings or more generally codewords to source symbols so that the code lengths correspond to the probabilities of occurrence of these symbols. This process guarantees that the most frequent symbols are matched with the shortest codes. This is used as a final stage of the data compression process, where the symbols and probabilities are provided by the part that does input modeling. Many encoding methods will be discussed later in this thesis.

In the more complex context of diffuse data, coding can involve many steps such as transforms and decompositions in order to reduce the size of the data. These steps allow to identify the characteristics of the data that need to be preserved by the compression scheme. Typically, entropy coding is applied subsequently to the result of these steps.

It is often the case that the decoding procedures are not discussed. It is justified since in most situations, the decoding process is obvious or can be easily derived from the coding process. However, we should always make sure that the decoding solutions provide the expected data. Moreover, the efficiency of the decoding algorithms is of more concern than that of the compression algorithm. For example, movies, photos, and audio data are often encoded once but decoded millions of times.

1.4 Classification of Compression Methods

Generally, compression techniques can be divided into two broad classes that almost overlap with the types of data we discussed previously. On one hand, we have the *lossless* compression methods in which we can recover the original data without any loss of information. On the other hand, there are the *lossy* compression schemes which do not guarantee to recover exactly the original data but can provide much higher compression rates.

We distinguish between the lossless algorithms, which can reconstruct the original information exactly from the compressed data, and lossy algorithms, which can only reconstruct an approximation of the original data or information. Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in the original information is often acceptable.

Other types of classification can also be done among compression methods. For instance, we can categorize the compression schemes as *static* or *dynamic*. A static method is one in which the mapping from the set of messages to the set of codewords is fixed before transmission begins, so that a given message is represented by the same codeword every time it appears in the source message. On the other hand, a code is dynamic if the mapping from the set of messages to the set of codewords changes over time. For example, typical dynamic compression methods involve computing an approximation to the probabilities of occurrence as the message is being transmitted. The assignment of codewords to message words is based on the values of the relative frequencies of occurrence at each point in time. Dynamic codes are also referred to in the literature as *adaptive*, in that they adapt to changes in the source data over time. The term *adaptive* is used for the remainder of this work.

Note that a third category can be created from the previous two. A compression method may be categorized as *hybrid* if it id neither completely static nor completely adaptive [12].

1.5 Performance Measures

If we have two or more methods of compression at hand, naturally we would like to compare them and discuss their relative merits. This can be done either by the complexity analysis of their algorithms, or by empirical tests and measurements of the quality and the amount of compression provided by each of them. Since the complexity analysis is a standard operation in computer science, we will focus here on the last two criteria which are the amount and quality of compression.

1.5.1 Compression Ratio

The amount of compression yielded by a coding scheme can be measured by a *compression ratio* C. Simply speaking, C can be defined as the ratio of the number of bits required to represent the data after compression to the number of bits required to represent the data before compression. However, the compression ration can be defined in several alternative ways. For example, the definition

$$C = \frac{\text{average codeword length}}{\text{average message length}}$$

captures the comparison of the average length of the coded message and the average length of the original source message. Another definition given by

$$C = \frac{S - O - OR}{S}$$

takes into account the representation of the code itself in the transmission cost. In this definition, S represents the length of the source message, O the length of the coded message, and OR the size of the output representation (the number of bits required to represent the coded message). The intention in this definition is to measure the total size of the transmission or the data to be stored [12].

1.5.2 Fidelity Criteria

In lossy compression schemes, the reconstructed data is different from the original one. In this context, it is important to measure the quality of compression in addition to its efficiency. The efficiency can still be measured with the compression ratio. One of the obvious measures of quality is the *distortion* which quantifies the difference between the original data and the reconstructed data.

Other measures of quality exist. They are typically based on the distortion and the ability to express the data in term of mathematical functions. For example, in the context of image data, we use *fidelity criteria* [18] which is expressed as the *root-mean-square* (`rms`) error between the original input image and the reconstructed image obtained after the compression and the decompression of the input image. For any value (x, y), which stand for the pixel coordinates, the distortion or error at (x, y) is given by

$$e(x, y) = \hat{f}(x, y) - f(x, y),$$

where $f(x, y)$ is the value of the original image at (x, y) and $\hat{f}(x, y)$ is the value of the compressed image at (x, y). Hence, the total error between the two images is

$$e_T = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} |e(x, y)|$$

where the images are of size $M \times N$. The *root-mean-square error* between the two images is expressed as

$$e_{rms} = \left[\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [e(x,y)]^2 \right]^{\frac{1}{2}}.$$

Another measure of quality derived from the *root-mean-square error* is called the *mean-square signal-to-noise ratio* and denoted SNR_{ms}. Assuming that $\hat{f}(x,y)$ is the sum of the original signal and the noise signal $e(x,y)$, we define

$$\text{SNR}_{ms} = \frac{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[\hat{f}(x,y) \right]^2}{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [e(x,y)]^2} \tag{1.4}$$

The `rms` value of the SNR is obtained by taking the square root of Equation 1.4.

The fidelity criteria expressed by a formula are called *objective fidelity criteria*. In the domain of imaging and video data, there are perceptual evaluations that can be done by the viewers, which are called *subjective fidelity criteria*. This can be done by recording a score obtained by presenting a decompressed image or video to a group of viewers and averaging their evaluations. The evaluations may be made for instance using an absolute rating scale.

Chapter 2

Classes of Sources and Data Modeling

In the previous chapter, we have defined a source as a finite set of symbols (the data) together with a probability distribution that attaches to each symbol a probability that is supposed to reflect the frequency of occurrence of that symbol in the context of the application where the data is used. A good model for the source data means finding the best probability distribution that captures the structure and the characteristics of the source data. In order to devise a model for source data, we need to examine the information we currently hold about the source. This includes what we know about the data's structure and composition, its worst case scenario for compression, and its best case scenario for compression. Good models for source data lead to more efficient compression algorithms. Indeed, compression techniques are mathematical operations that can be optimized when a good mathematical model is available for the source. There are several approaches to building mathematical models.

2.1 Types of Data and Models

2.1.1 Physical and Structural Models

If we happen to know something about the physical laws that were used to generate the data, then we can use that information to construct a model. For example, in imaging and video applications, the knowledge about the physics of the image formation process can be used to construct a mathematical model for the sampled image and video data. However, in most applications the physics of data generation is simply unknown or too complicated to understand. In such case, we recourse to designing models based on empirical observations of the statistics of the data. Ideally, we would know that in a particular case we have, e.g., a Jules Verne novel, a Monet painting, or an article from the Gazette for which patterns and statistics are available.

For such an approach to be possible, we need to provide modeling methods for all writers, painters, newspapers, and so on. The number of modeling methods would be incredibly large in this case. Moreover, we do not know the future writers and painters and we cannot prepare models for their works and matching their styles. On the other hand, if no assumption is made about the characteristics of a given source of data, we would need to treat everything as a general case, and we have no way of finding similarities in the data and taking advantage of some assumed properties.

In order to achieve efficient compression procedures, we have to make a compromise. The standard approach is to define a reasonable number of classes of sources that produce sequences of data of different types. When dealing with particular data, we assume that it can be treated as an output of one or a combination of the predefined sources. Then, we choose the source's class which approximates the data best. Consequently, we apply appropriate compression algorithms that work well on the source from the chosen class. This approach allows to reduce the number of modeling methods to a reasonable level.

2.1.2 Simple Probability Models

The simplest probability model for a source is to assume that every symbol emitted by the source is independent of every other symbol, and each occurs with the same probability. More formally, if the source has n symbols $\{s_1, \ldots, s_n\}$, the emission (or choice) of the symbol s_i can be seen as a random variable X_i in such a way that the sequence of random variables $X_1, \ldots, X_n$ are independent and identically distributed (**i.i.d.**). This model is sometimes labeled as the *ignorance model*, as it doesn't allow to make any useful predictions about which symbols may appear in the future based on what is emitted by the source now. This model will generally correspond to the case when we know nothing about the source and it yields the highest entropy for the source with symbols $\{s_1, \ldots, s_n\}$ among all the possible probability models. In a sense, this corresponds to the worst case scenario as it involves high uncertainty and complexity in the data. This type of data has low symbol repetition and it is characterized by high source entropy.

The next step up in the probability modeling is to keep the independence assumption while removing the equal probability assumption. A discrete probability distribution is assigned to the source, where the probability of an individual symbol reflects the frequency of occurrence of that symbol in the emission process. The best case scenario would be that most of the probabilities are zero and the remaining ones are high (the sum of all probabilities should be equal to 1). This case involves highly predictable data. A data sequence with highly repetitive data would be characteristic of these sources. This is characterized by low source entropy.

In many applications, the assumption of independence does not fit with the structure of the data. For instance, for image and video data, there is temporal, spatial, and color space redundancy in the data. Likewise, in text and language data, there is redundancy and correlation between observed letters and words. It is therefore desirable to discard the independence assumption. We need to come up with probability models that can describe the source data where symbols are not necessarily independent from each other. There can be all sort of dependencies between data, and it is hard to represent that accurately with any model. The common practice is to use models that express special form of dependencies. One of the most popular ways of representing dependence in the data is through the use of Markov models, named after the Russian mathematician Andrei Andrevich Markov (1856-1922).

2.1.3 Markov Model

In the domain of lossless compression, it is common to use a specific type of Markov process called a *discrete time Markov chain*. Let $\{x_n\}$ be a sequence of observations (may be infinite) over a finite set of symbols (alphabet) $\{s_1, \ldots, s_m\}$. This sequence is said to follow an order k Markov model if the probability of each observation only depends on the k previous observations, that is

$$P(x_n|x_{n-1}, \ldots, x_{n-k}) = P(x_n|x_{n-1}, \ldots, x_{n-k}, \ldots), \tag{2.1}$$

where x_i is the i-th observation generated by the source. This means that the knowledge of the past k symbols is equivalent to the knowledge of the entire past history of the process.

The most commonly model used in practice is the first-order Markov model, where the probability of observation x_n only depends on the previous observation x_{n-1}, that is

$$P(x_n|x_{n-1}) = P(x_n|x_{n-1}, \ldots, x_{n-k}, \ldots). \tag{2.2}$$

The process is then described using a set of transition probabilities

$$p_{ij} = P(X_n = s_j|X_{n-1} = s_i).$$

These denote the probability of symbol s_i being followed by symbol s_j. The probability transition matrix for the 1-st order transitions probabilities is given by

$$\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \ldots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mm} \end{bmatrix}$$

The conditional probabilities in Equations 2.1 and 2.2 only indicate the existence of dependance between observations. What we need in practice is to give explicitly the form of dependence between them. A panoply of models are developed using the first order Markov models and specific relationships in terms of mathematical formulae between the observations. For instance, we may assume that the dependence between two consecutive observations is expressed as a linear function, then we would have an equation of the type

$$x_n = \rho x_{n-1} + \xi_n \tag{2.3}$$

where ξ_n is a stationary random process with 0 mean, typically referred to as white noise process. This type of model is often used when developing coding algorithms for audio and image data. Note that the use of Markov model does not require in any way the existence of linear relationship between observations.

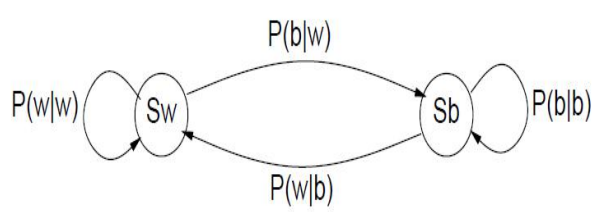

Figure 2.1: A two state first-order Markov Model

This is the case if we consider the discrete time Markov chain representing the probabilities that the source generates a black (b) or white (w) pixel in a binary image. We know that the appearance of a white pixel as the next observation depends on whether the current pixel is white or black.

The model is represented by the state diagram shown in Figure 2.1. Each node of the diagram represents one of the states, S_w and S_b, where S_w correspond to the case where the current pixel is a white pixel, and S_b corresponds to the case where the current pixel is a black pixel. Each edge in the diagram represents a conditional probability (transition probability) of generating a particular pixel. For example, $P(w|b)$ is the conditional probability of generating a white pixel given that the previous one was black, and $P(b|w)$ has a similar interpretation. We define the probability of being in each state as $P(S_w)$ and $P(S_b)$.

2.2 Classes of Sources

2.2.1 Memoryless Sources

A discrete memoryless source (DMS) is defined by the property that its output at a certain time does not depend on its output at any earlier time. Hence, the parameters of a memoryless source do not depend on the number of symbols it generated so far. This means that the probability of occurrence of each symbol is independent from its position in the output sequence. Typically, these sources are characterized by a set of symbols $\{s_1, \ldots, s_n\}$, and associated probabilities $\{p_1, \ldots, p_n\}$, such that the frequency of symbol occurrences are close to the corresponding probabilities. A DSM source can be viewed as finite-state machine (FSM) with a single state and n loop-transitions. Each transition is denoted by a different character, s_i, from the alphabet, and with each of them the probability p_i is associated. An example of a memoryless source with alphabet $\{a, b, c, d, e\}$ and with associated probabilities $\{0.4, 0.2, 0.1, 0.1, 0.2\}$ is shown in Figure 2.2.

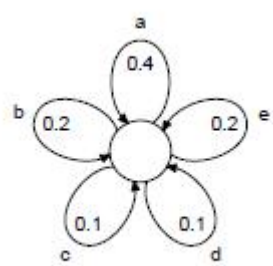

Figure 2.2: Example of a memoryless source.

2.2.2 Markov Sources

A Markov source is a source that has a memory of a limited number of steps back in time. We distinguish between the Markov sources that have a memory of one symbol that we call order-1 Markov sources or just Markov sources and the Markov sources that have a memory of k symbols ($k > 1$) called order-k Markov sources. An order-k Markov source can be modeled as a finite-state machine. There can be at most k transitions from each state and all of them are labeled with a different symbol. Each transition has some associated probability and the transition probabilities for each state (outgoing transitions) must sum to 1. The current state is completely specified by the last k symbols observed.

Figure 2.3 shows an order-1 Markov source with the alphabet $\mathcal{A} = \{a, b, c\}$. This source is a FSM with three states labeled as a, b, and c. The transition probabilities $P(x_n|x_{n-1})$ between the three states are shown as labels of the edges between them.

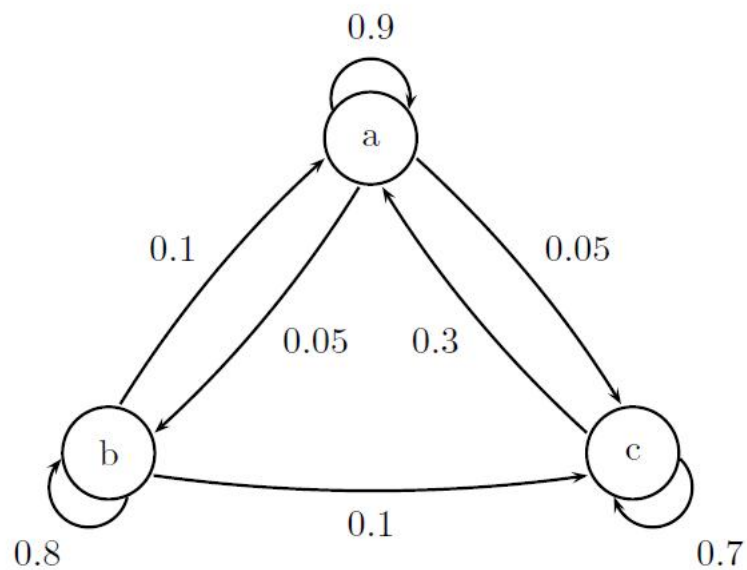

Figure 2.3: Example of an Order-1 Markov source.

Figure 2.4 shows an order-2 Markov source with the alphabet $\mathcal{A} = \{a, b\}$. This source is a FSM with four states labeled as aa, ab, ba and bb. The transition probabilities $P(x_n|x_{n-1}, x_{n-2})$ (used as edge labels) are given by

	a	b
aa	$P(a\|aa) = 0.9$	$P(b\|aa) = 0.1$
ab	$P(a\|ab) = 0.7$	$P(b\|ab) = 0.3$
ba	$P(a\|ba) = 0.2$	$P(b\|ba) = 0.8$
bb	$P(a\|bb) = 0.05$	$P(b\|bb) = 0.95$

Note that the transition probabilities for each state must sum to 1.

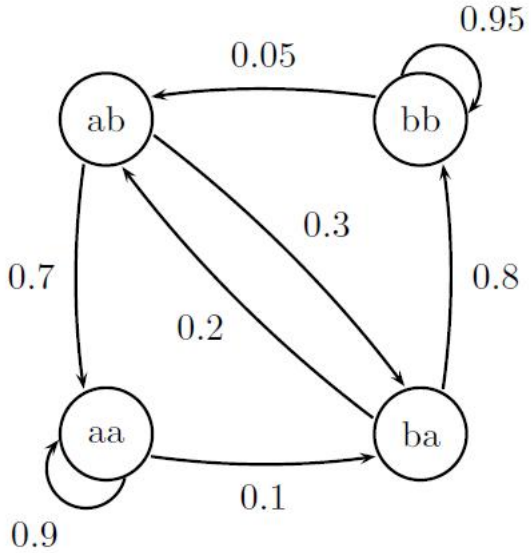

Figure 2.4: Example of an order-2 Markov source.

Memoryless and Markov models were first introduced in the original work of Shannon [9]. Shannon used the Markov models specifically for English text compression. Since then, the use of Markov models in text compression has become a rich and active area of research [19]. In this work, we focused on the most popular source models which happen to be the ones that are used the most in practice. There is a panoply of other source models that are not discussed here such as the FSMX sources [20] and the context tree sources [21].

Chapter 3

Static Huffman and Arithmetic Methods

In this chapter, two classical methods for data compression are discussed. These methods are the Huffman and the Arithmetic coding methods. First section of this chapter discusses the Huffman coding and different methods to perform the Huffman compression. Limitations of Huffman coding are also discussed in this section. Second section of the chapter presents another classical method, the Arithmetic coding. Different processes involved in Arithmetic coding and decoding are discussed.

3.1 Huffman Compression Method

The Huffman coding method is the best known form of entropy coding [12]. The basic idea is to assign codewords from a prefix code to symbols according to the probabilities with which they appear in the message ensemble, in such a way that symbols occurring more frequently are represented by short codewords, while messages with smaller probabilities map to longer codewords. This is achieved by using a particular algorithm called the Huffman Coding Algorithm developed by David Huffman as a student in a class on information theory at MIT in 1950 [8]. The Huffman coding is naturally uniquely decodable and no delimiters or extra information is inserted in the code words to facilitate decodability. This is a direct consequence of using prefix codes.

The Huffman coding method is devised from two important observations related to optimum prefix codes as stated in the following Lemma. Here the notion of optimum code is defined as the code with minimum redundancy. As stated in Subsection 1.2.4, for a source with a fixed probability distribution a code is optimal if it has the minimum average codeword length.

Lemma 3.1. *Let $\mathcal{S} = (S, P)$ be a source, where $S = \{s_1, \ldots, s_n\}$ and (C, f) be a coding scheme, where $C = \{c_1, c_2, \ldots, c_n\}$ is a prefix code. Then we have the following two assertions.*

1. *If C is an optimum code, then the symbols that occur more frequently must be assigned shorter codewords than symbols that occur less frequently.*

2. *If C is an optimum code, then the two symbols that have the lowest frequency of occurrence must be assigned codewords that have equal length.*

Proof. First, if the symbols with high occurrence frequencies are assigned longest codewords, then the average codeword length

$$\sum_{i=1}^{n} P(s_i)\texttt{len}(f(s_i))$$

would be larger than if the shortest codewords were assigned. This is immediate from the formula above as large values of $P(s_i)$ will be associated with large values of $\texttt{len}(f(s_i))$. This will result in a smaller compression ratio then if the situation is reversed. Thus, a code that assigns longer codewords to symbols that occur more frequently cannot be optimum. In order to prove the second assertion, assume that C is an optimum code in which the two codewords c_i and c_j corresponding to the two least frequent symbols do not have the same length. Assume further, without loss of generality, that

$$\texttt{len}(c_j) = \texttt{len}(c_i) + k.$$

Since we are dealing with a prefix code, the shorter codeword c_i cannot be a prefix of the longer codeword c_j. It follows that even if we drop the last k bits from c_j and obtain a new codeword c'_j, c_i and c'_j would still be distinct. Note that since c_i and c_j are assigned to the least probable symbols in the alphabet, no other codeword can be longer than c_i. Hence, c'_j cannot become the prefix of some other codeword. It follows that if we replace c_j with c'_j, we will obtain a new code C' that has a shorter average codeword length than C. This is a clear contradiction since we assumed that C is an optimal code. Therefore, for an optimal code the second assertion also holds true.

■

Huffman's algorithm is very simple and is most easily described graphically in terms of how it generates a prefix-code tree. The algorithm takes as input a list of positive weights

$$L := [w_1, \ldots, w_n]$$

and constructs a full binary tree whose leaves are labeled with the weights. We recall that a binary tree is full if every node has either zero or two children. The weights $w_i = p_i$ represent the probabilities associated with the source symbols.

The outline of the algorithm is as follows.

Algorithm 1 [Huffman's Algorithm]

```
Start with a set of singleton trees, one for each symbol. Each tree contains a single vertex with
    weight w_i.
while there is more than one tree in the structure do
    Select two trees with the smallest weight roots w_i and w_j.
    Merge them into a new tree with a root containing the weight w_i + w_j, and making the two
    trees its children.
    Remove the weights w_i and w_j from the list, and insert the weight w_i + w_j
end while
Perform the tree traversal and for each parent node label the edge to its left child with the digit
    0 and the edge to the right child with 1.
for each source symbol s do
    Concatenate the labels along the path from the root to the leaf node representing s.
    Output the sequence of labels as the codeword for the symbol s
end for
```

Complexity analysis

First, note that at each step of the loop in Algorithm 1, it does not matter what we choose as the left or right child, but the convention will be to put the lower weight root on the left if $w_i \neq w_j$. This process continues until the weight list contains a single value, which is equivalent to having a single tree in the structure. Note also that it is desirable to start the process with a decreasing list of weights and maintain the list in that state at each step. Although this is not important to the correctness of the algorithm, it does provide a more efficient and easy implementation of the algorithm.

For a source symbol of size n, this algorithm will require at most $n-1$ steps to process all the symbols. This stems for the fact that every complete binary tree with n leaves has $n-1$ internal nodes, and each step creates one internal node. If we use a priority queue with $O(\log n)$ time insertions and find-mins, it is easily seen that the algorithm will run in $O(n \log n)$ time.

Example 3.2. Let us consider a source $\mathcal{S} = (S, P)$, with alphabet $S = \{s_1, s_2, s_3, s_4, s_5\}$ and with probability distribution P given by

$$P(s_2) = 0.4, P(s_4) = 0.2, P(s_1) = P(s_3) = 0.15, \text{ and } \quad P(s_5) = 0.1.$$

The binary entropy for this source, which conveys the average amount of information contained in the source, can be calculated using the formula

$$H(\mathcal{S}) = -\sum_{i=1}^{5} P(s_i) \log_2 P(s_i),$$

which is found to be equal to 2.15. In theory, this means we need on average, at least, 2.15 bits per symbol to encode the information generated by the source $\mathcal{S}$. We will now design the Huffman code of the source $\mathcal{S}$ by following the steps of Huffman's algorithm discussed above.

1. First, we sort the symbols $\{s_1, \ldots, s_5\}$ in descending or ascending order of their associated probability values. For this example the symbols are arranged in descending order as shown in the table below.

Symbol	Probability
s_2	0.4
s_4	0.2
s_1	0.15
s_3	0.15
s_5	0.1

2. We show the Huffman's algorithm on the list of sorted values using the following diagram:

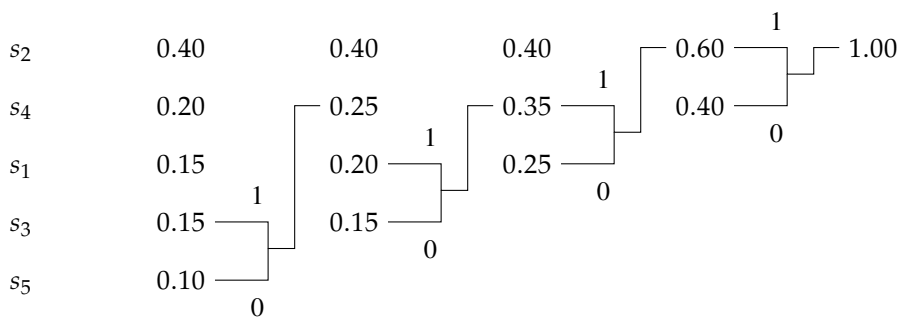

Figure 3.1: Huffman algorithm shown on the list.

3. We can also visualize the steps of Huffman's algorithm using a binary tree. Note that the leaves pictured as rectangles represent the symbols while the internal nodes pictured as circles are nodes that contain the sum of the weights of their corresponding children. Whenever two leaves are combined, they form an internal node. The internal nodes in turn combine with other nodes or leaves to form more internal nodes. The leaves and nodes are ordered in increasing order from left to right.

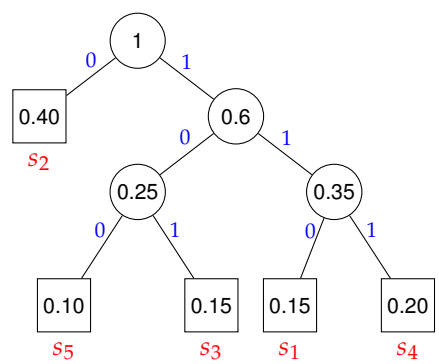

Figure 3.2: Code tree for the list of symbols.

4. From the tree, we read the Huffman code and record it in the following table

Symbol	Probability	Codeword	Length (ℓ_i)
s_1	0.15	110	3
s_2	0.4	0	1
s_3	0.15	101	3
s_4	0.2	111	3
s_5	0.1	100	3

5. The average codeword length for the Huffman code provided here is bits per symbol.

6. It follows that the redundancy of this Huffman code is given by

$$R(C) = \overline{\ell} - H(\mathcal{S}) = 2.2 - 2.15 = 0.05 \quad \text{bits/symbol.}$$

3.1.1 Optimality and Length of Huffman Codes

In Lemma 3.1, we have proved that the following two conditions are satisfied by any optimal variable length code.

1. Given any two source symbols s_i and s_j, if $P(s_j) > P(s_i)$ then $\ell_j < \ell_i$, where ℓ_k is the length of the codeword s_k.

2. The two of the least probable symbols have codewords with the same maximum length.

This simply means that these are necessary conditions for a code to be optimal. There are two more necessary conditions that are related to the structure of binary tree used in the Huffman algorithm.

3. In the tree corresponding to the optimum code, there must be two branches stemming from each intermediate node.

4. If we change an intermediate node into a leaf node by combining all the leaves descending from it into a composite word of a reduced alphabet, then, if the original tree was optimal for the original alphabet, the reduced tree is also optimal for the reduced alphabet.

A formal proof of these last two conditions can be found in Fano [22]. We discuss here an intuitive justification of these two assertions. In assertion 3, if there were any intermediate node with only one branch coming from that node, we could remove it without changing the nature of the code while reducing its average codeword length. This is clearly a contradiction to the assumption that the code is optimal. If the condition in assertion 4 is not satisfied, we could find a code with smaller average codeword length for the reduced alphabet and then simply expand the composite word again to get a new code tree that would have a shorter average codeword length than the original optimal code. This would contradict our statement about the optimality of the original code.

It remains now to show that if the four assertions above are satisfied, then the corresponding code is optimal. Again the general formal proof can be found in Fano [22]. If conditions 1, 2, 3, and 4 are satisfied, the the two least probable source symbols would have to be assigned codewords of maximum length ℓ_m. Furthermore, the leaves corresponding to these letters arise from the same intermediate node. This is equivalent to saying that the codewords for these letters are identical except for the last bit. Consider the common prefix as the codeword for the composite letter of a reduced alphabet. Since the code for the reduced alphabet needs to be optimal for the code of the original alphabet to be optimal, we can continue the same procedure again and iterate it until we have a reduced alphabet of size one. This corresponds exactly to the Huffman algorithm. Therefore, the necessary conditions above, which are all satisfied by the Huffman procedure, are also sufficient conditions.

At this point, we would like to know whether it is possible to quantify the average codeword length of a Huffman code. We will show that the Huffman code for a source $\mathcal{S}$ has an average code length $\overline{\ell}$ bounded below by the entropy and bounded above by the entropy plus 1 bit, that is

$$H(\mathcal{S}) \leq \overline{\ell} < H(\mathcal{S}) + 1. \tag{3.1}$$

This result is based on the Kraft and McMillan result introduced in Chapter 1. The first part of the result asserts that for a uniquely decodable code C with K codewords, the lengths $\{\ell_i\}_{i=1}^K$ of the codewords must satisfy the inequality

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

The second part states that for a sequence of positive integers $\{\ell_i\}_{i=1}^K$ satisfying the inequality above, there must exist a uniquely decodable code whose codeword lengths are given by the sequence $\{\ell_i\}_{i=1}^K$. We also recall that for a source $\mathcal{S}$ with alphabet $S = \{s_1, \ldots, s_K\}$, and probability model $\{P(s_1), \ldots, P(s_n)\}$, the average codeword length is given by

$$\overline{\ell} = \sum_{i=1}^{K} P(s_i)\ell_i.$$

We can now write

$$\begin{aligned}
H(\mathcal{S}) - \overline{\ell} &= \sum_{i=1}^{K} P(s_i) \log_2 \frac{1}{P(s_i)} - \sum_{i=1}^{K} P(s_i)\ell_i \\
&= \sum_{i=1}^{K} P(s_i) \left[\log_2 \frac{1}{P(s_i)} - \ell_i\right] \\
&= \sum_{i=1}^{K} P(s_i) \left[\log_2 \frac{1}{P(s_i)} - \log_2 2^{\ell_i}\right] \\
&= \sum_{i=1}^{K} P(s_i) \log_2 \left(\frac{2^{-\ell_i}}{P(s_i)}\right) \\
&\leq \log_2 \sum_{i=1}^{K} 2^{-\ell_i} \\
&\leq 0.
\end{aligned}$$

The last inequality comes directly from the Kraft-MacMillan inequality, i.e. $\sum_{i=1}^{K} 2^{-\ell_i} \leq 1$, while the inequality before it is a well known result in probability and mathematical analysis, called Jensen's inequality. It states that if a function $f(x)$ is concave, then $E((f(X)) \leq f(E(X))$, where, the notation $E(X)$ stands for the expected value of the random variable X. Obviously, the logarithm is a concave function on its domain, hence $H(\mathcal{S}) \leq \overline{\ell}$ for any optimal code.

Observe that for each $1 \leq i \leq K$, we can select a value ℓ_i' such that

$$\log_2 \frac{1}{P(s_i)} \leq \ell_i' < \log_2 \frac{1}{P(s_i)} + 1.$$

It is easy to see that such ℓ_i' will satisfy $2^{-\ell_i'} \leq P(s_i)$, and hence

$$\sum_{i=1}^{K} 2^{-\ell_i'} \leq \sum_{i=1}^{K} P(s_i) = 1.$$

The Kraft and McMillan result guarantees the existence of a uniquely decodable code C with codeword lengths $\{\ell'_i\}$. The average length of C satisfies

$$\sum_{i=1}^{K} P(s_i) \log_2 \frac{1}{P(s_i)} \leq \sum_{i=1}^{K} P(s_i)\ell'_i < \sum_{i=1}^{K} P(s_i) \left(\log_2 \frac{1}{P(s_i)} + 1 \right),$$

that is

$$H(\mathcal{S}) \leq \overline{\ell'} < H(\mathcal{S}) + 1.$$

Since Huffman code is optimal, its average codeword length $\overline{\ell}$ must satisfy

$$H(\mathcal{S}) \leq \overline{\ell} \leq \overline{\ell'} < H(\mathcal{S}) + 1.$$

Note that tighter bounds can be derived for optimal codes (see [23]).

3.2 Arithmetic Coding

In the previous section, we discussed Huffman coding which produces the best codes for the individual source symbols. However, in the context of binary strings codewords and the binary tree structure, we can easily see that the only case where Huffman coding produces an ideal variable-length code is when the symbols have probabilities of occurrence that are negative powers of 2. This is because the Huffman method assigns a code with an integer number of bits to each symbol in the alphabet. For example, a source symbol with probability 0.4 should ideally be assigned a 1.32-bit code, since its entropy is $\log_2 0.4 = 1.32$. However, the Huffman method will normally assign to the same symbol a code of 2 bits.

Arithmetic coding overcomes the problem of Huffman codes by assigning one code to an entire input sequence of symbols. Starting with an interval on the real number line, typically $[0, 1)$, the method reads the input sequence one symbol at a time, and uses the probability of each symbol to narrow the interval. It is quite easy to grasp that a narrow interval requires more bits to be specified than a longer one. For instance, using the lower and upper limits or by one limit and width, the interval $[0, 1)$ can be specified by the two 1-bit numbers 0 and 1 whereas the interval $[0.12575, 0.1257586)$ will clearly require longer bit binary numbers to be specified. For an efficient compression, the algorithm is designed in such a way that a high-probability symbol narrows the interval less than a low-probability symbol. Thus, high-probability symbols will contribute fewer bits to the output code.

Arithmetic coding came from Shannon's observation that sequences of symbols can be coded by their cumulative probability and it was presented by Abramson [24] in his text on information theory.

3.2.1 Steps of Arithmetic Coding

The main idea of arithmetic coding is to represent each possible sequence of m symbols by a separate interval on the number line between 0 and 1. For a sequence of symbols $\alpha = \alpha_1\alpha_2 \ldots \alpha_m$ with probabilities $\pi_1, \ldots, \pi_m$, the arithmetic coding algorithm will assign a sub-interval of $[0, 1)$ of size

$$\prod_{i=1}^{m} \pi_i,$$

by starting with the interval $[0, 1)$ and narrowing it successively by a factor of π_i for each symbol α_i of the sequence.

The method begins with a source $\mathcal{S} = (S, P)$, where $S = \{s_1, \ldots, s_n\}$ is a set of symbols or alphabet, and $P = \{p_1, \ldots, p_n\}$ is the associated probability distribution. Initially, the interval $[0, 1)$ is partitioned into subintervals on the basis of cumulative probabilities. The cumulative probability function (cpf) $f(x)$ of a probability distribution $P = \{p_1, \ldots, p_n\}$ is given by

$$f(k) = \sum_{i=1}^{k} p_i, \text{ for } k = 0, \ldots, n.$$

Note that $f(0) = 0$ and $f(n) = 1$. The partition of $[0, 1)$ is given by $\{[f(k), f(k+1))\}_{k=0}^{n-1}$, where symbol s_k corresponds to the sub-interval or the range $[f(k-1), f(k))$ in the partition.

Narrowing Procedure

The procedure for generating the sub-interval I_α associated with the sequence α works inductively as follows.

1. We define $I_0 = [0, 1)$ as the initial interval in the sequence. I_0 is partitioned as discussed above.

2. The appearance of the first symbol α_1 in the sequence $\alpha = \alpha_1\alpha_2 \ldots \alpha_m$ allows to find the next element in the sequence, i.e. I_1. Since α_1 is necessarily equal to one of the source symbols, say s_i, then $I_1 = [f(i-1), f(i))$.

3. The subinterval I_1 is now partitioned in exactly the same proportions as I_0, into a family of sub-intervals corresponding to the symbols $s_1, \ldots, s_n$. It is easily established that the j-th sub-interval corresponding to the symbol s_j in the new partition is given by

$$[(f(i) - f(i-1))f(j-1) + f(i-1), (f(i) - f(i-1))f(j) + f(i-1)).$$

- If the second symbol α_2 in the sequence α is equal to s_j, then

$$I_2 = [(f(i) - f(i-1))f(j-1) + f(i-1), (f(i) - f(i-1))f(j) + f(i-1)).$$

  Note that $f(i) - f(i-1) = p_i$, $f(i-1)$ is the left endpoint or limit of the interval I_1, and $f(j-1)$ is the left endpoint of the interval associated with s_j in the partition of I_1. Moreover, the length of I_2 is equal to

$$(f(i) - f(i-1))(f(j) - f(j-1)) = p_i \times p_j,$$

- If we parameterize our intervals I_i corresponding to the appearance of each symbol α_i with the left endpoint L_i and the length ℓ_i, then we can write a recurrent formula to describe the narrowing procedure:

$$\begin{cases} \ell_i &= \ell_{i-1} \times \pi_i \\ L_i &= L_{i-1} + \ell_{i-1} \times l_{\alpha_i}, \quad i \geq 1. \end{cases} \tag{3.2}$$

- Note that $\ell_0 = 1$ and $L_0 = 0$ as they correspond to the interval $I_0 = [0, 1)$. The notation l_{α_i} stands for the left endpoint of the interval associated to α_i in the subdivision of I_0. As stated earlier, if $\alpha_i = s_j$, then $l_{\alpha_i} = f(j-1)$.

- We can easily prove using the sequence of partitions that the final subinterval obtained with one sequence is disjoint from any other subinterval that may have been generated using this process for any other sequence.

4. When the final subinterval is reached, a number called a **Tag** is chosen. It can be any point of that interval. The obtained tag can be converted into binary and the bits are used as the code.

5. It can be easily proven by induction that the size of the final interval I_m is

$$\ell_m = \pi_1 \times \pi_2 \times \ldots \times \pi_m.$$

We will illustrate the steps of the arithmetic coding in the following example which uses the source symbols and the probability model in Example 3.2.

Example 3.3.

1. The corresponding cumulative probabilities and the range in the partitioning of $I_0 = [0, 1)$ for each symbol are shown in the following table:

Symbol	Probability	Cumulative probability	Range
s_1	0.15	0.15	[0, 0.15)
s_2	0.4	0.55	[0.15, 0.55)
s_3	0.15	0.70	[0.55, 0.70)
s_4	0.2	0.90	[0.70, 0.90)
s_5	0.1	1.00	[0.90, 1.00)

2. We will assume for simplicity that the message is $\alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$, where $\alpha_i = s_i$. Hence, $I_0 = [0, 1)$ and $I_1 = [0, 0.15)$ since $\alpha_1 = s_1$. It i also in accordance with the recurrence formula as $\ell_1 = \ell_0 \times \pi_1 = 1 \times 0.15 = 0.15$ and $L_1 = L_0 + \ell_0 \times l_{\alpha_1}$. We know $L_0 = 0$ and $\ell_0 = 1$. Since t$\alpha_1 = s_1$, then $_{\alpha_1} = f(0) = 0$. Hence $L_1 = 0$.

3. We can calculate the subsequent partitions as explained in the beginning of this section and continue building recursively the coding interval by performing subsequent subdivisions corresponding to the remaining symbols of the sequence as shown in the Figure 3.3.

4. The overall arithmetic encoding process for the message $\alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$ is shown in and the final interval is given by $I_5 = [0.06342, 0.06360)$.

- Note that the size of I_5 is $\ell_5 = 0.06360 - 0.0342 = 0.00018$, which is also equal to the product of the initial probabilities

$$\ell_5 = p_1p_2p_3p_4p_5 = 0.15 \times 0.4 \times 0.15 \times 0.2 \times 0.1 = 0.00018.$$

- A value in I_5, i.e. between 0.06342 and 0.06360 can be generated and assigned as a code to the sequence α. Typically, we use the midpoint of the interval, that is the average value of its endpoints which can be converted to binary to get a binary code. For this case, the average value is 0.06351.

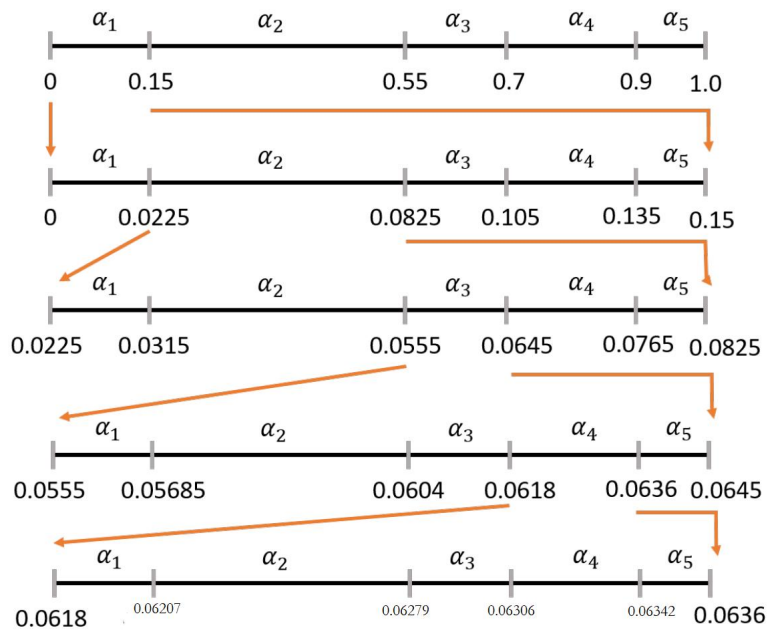

Figure 3.3: Arithmetic coding for symbol message $\alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$

3.2.2 Steps for Arithmetic Decoding

In order to recover an encoded message using the arithmetic coding method, we need to have at our disposal the model of the source used by the encoder and the tag within the interval determined by the encoder. The decoding process is quite similar to the encoding process and consists of a series of comparisons of the tag to the ranges representing the source messages.

Formally, the decoding requires the following input:

- Encoded Value
- Frequency table or probability of the symbols
- Number of symbols in the message

The decoding process consists of several steps as discussed below:

1. First step is to calculate the probability of each symbols in the message and build the table. After that the decoding process is similar to the encoding. Start with the decoding line as shown in Figure 3.4.

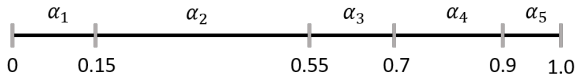

Figure 3.4: Arithmetic Decoding Step 1.

2. Search for the encoded value interval. In the example the tag is 0.06351 and it lies between 0 and 0.15. So the first symbol should be α_1.

3. We restrict the sub-interval to $[0, 0.15)$ and update the intervals just like the encoding. This process is presented in Figure 3.5.

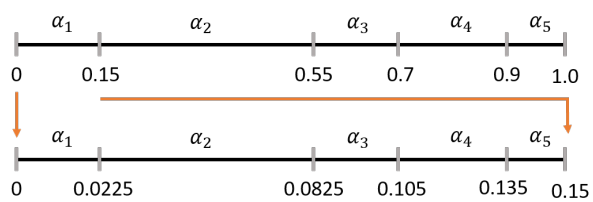

Figure 3.5: Arithmetic Decoding Step 2.

Now, the tag 0.06351 lies between 0.0225 and 0.0825, so the second symbol is α_2.

4. Repeat the previous step until the decoded symbols are equal to the number of encoded symbols. The overall process is shown in Figure 3.6.

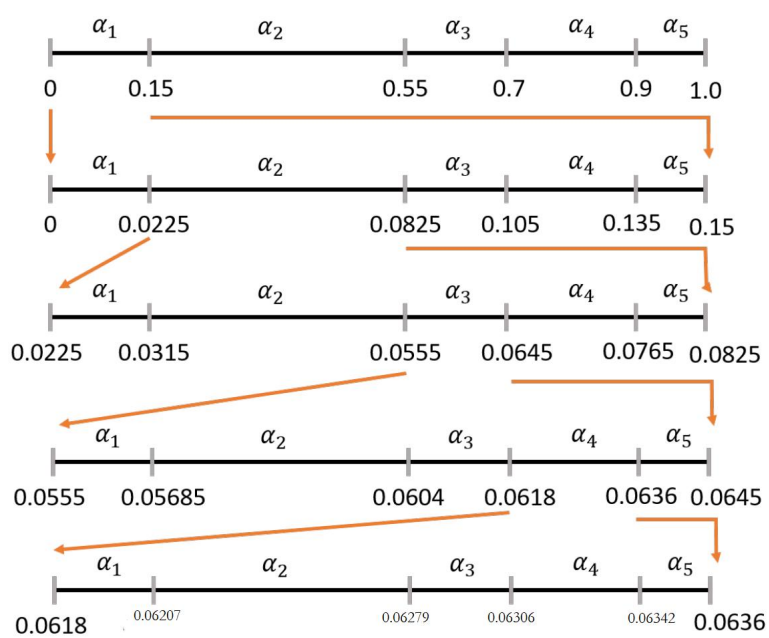

Figure 3.6: Arithmetic decoding of a message with tag 0.06351 and source data give in Example 3.2.

- We can see now from the figure the decoded message is $\alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$.

3.2.3 Properties of Arithmetic Coding

We reiterate that the final subinterval obtained with one sequence is disjoint from any other subinterval that may have been generated using this process for any other sequence. There is a question wether the uniqueness remains valid when the code is converted into binary code. There are many works (see [1, 13]) that prove this is still the case and we can provide from tags or the intervals obtained in the arithmetic coding a uniquely decodable binary codes.

Arithmetic coding has many advantages. First, unlike Huffman coding, no code tree needs to be transmitted to the receiver. Encoding is done for a group of symbols and not for each symbol separately. This leads to higher compression ratios. The final advantage of Arithmetic coding is its use of fractional values. In Huffman coding there is some code waste. It is only optimal for coding symbols with probabilities that are negative powers of 2. Huffman coding will rarely reach optimality in real data because the entropy takes rarely integer values.

One disadvantage of Arithmetic coding is its complex operations. Arithmetic coding consists of many additions, subtractions, multiplications, and divisions. It is difficult to create efficient implementations for Arithmetic coding. These operations make Arithmetic coding significantly slower than Huffman coding. Another disadvantage consists of the precision problem. Encoding long messages requires many levels of subdivisions which leads to very tiny intervals which requires high precision operations on numbers. Fortunately, there are many implementations that address these issues [1, 13].

Chapter 4

Adaptive Coding

In most practical compression applications, the probability models are not known in advance. In that case, scanning of all the data is needed to provide accurate probabilities in order to perform the the Huffman and Arithmetic coding discussed previously. In some instances this may be an immense amount of data or the data may not be available entirely. Adaptive coding schemes were created to deal with this type of problem. In these schemes, the probabilities are estimated and updated as more input is processed. The designed code is adaptive. It is changing so as to remain optimal for the current estimates. In essence, the encoder is *learning* the characteristics of the source. The decoder must also learn along with the encoder as to stay in synchronization with the encoder. The main advantage of these systems is that they require only one pass over the data instead of two passes in the static coding. We will examine the adaptive Huffman and adaptive Arithmetic codings.

4.1 Adaptive Huffman coding

The adpative Huffman coding was introduced independently by Faller [25] and Gallager [23]. Knuth [26] improved the original algorithm, and the resulting algorithm is referred to as algorithm FGK. There is a number of different versions of the adaptive Huffman coding and two of the most famous methods are Faller-Gallager-Knuth (FGK) and Vitter (V) algorithms [27]. The major difference between the two algorithms is that in FGK the tree size increases with each new symbol and it does not check the previous nodes while updating the tree. On the other hand, the V algorithm keeps the size of the tree to a minimum by checking previous nodes while updating the tree.

The adaptive Huffman algorithm is based on the sibling property according to which, each node of the Huffman tree has a sibling except the root node, and the nodes can be listed in order of decreasing weight with each node adjacent to its sibling. In order to fulfill this property, track of each node's weight and order is kept. Here, order is used to number the nodes according to the weight and it decreases from right to left and bottom to top. On the other hand, weights are used to keep the frequency of the symbol. We have chosen to present Algorithm V in the following since it is an improved version of the Algorithm FGK.

In both FGK and V algorithms, the sender and the receiver start with the same tree structure and maintain it dynamically to form the same changing Huffman code trees. Initially, the code tree at both the sender and the receiver consists of a single leaf node, called the 0-node. It corresponds to all symbols *not yet transmitted* (NYT) and has a weight of zero. Before the beginning of transmission, a fixed code for each symbol is agreed upon between the sender and the receiver. As transmission

progresses, nodes corresponding to symbols transmitted will be added to the tree, and the tree is reconfigured using an update procedure.

Algorithm V Requirements

For the adaptive Huffman coding, the code tree must satisfy the following:

- All the nodes except the root have siblings.
- Each node is assigned a unique number from 1 to $2n - 1$ (if we have n symbols)
- From top to bottom, nodes are ordered according to their weights. A node with higher weight will have higher order.
- On each level, from left to right the order increases.
- On each level, the nodes may or may not have the same weights. However, the nodes with same weights will have consecutive order number.
- All leaf nodes except the Not Yet Transmitted (NYT) node contain symbol values.
- NYT is the leaf node where all new symbols are added.
- The weight of a leaf node is equal to the number of times the symbol in the leaf has been encountered.
- The weight of any internal node of the tree is equal to the sum of the weights of its child nodes.

When a symbol is read, it is checked in the tree and if it is already available then its weight and order are adjusted. If it is a new symbol, then it is added to the NYT and the tree is updated accordingly.

4.1.1 Main Steps of the Algorithm

1. Initialize the tree with one node (root) with weight $w_r = 0$
2. Based on the new symbol (S), the tree is updated as follows:
   (a) If pointer P for the new symbol S points to NYT
      i. Extend the tree by adding two new nodes
      ii. Right node will be the leaf corresponding to symbol S
      iii. Left node will become the new NYT node
      iv. Update weight of the right child of P
   (b) If pointer P for the new symbol S does not point to NYT
      i. Update P with the previous node corresponding to symbol S
      ii. If P is a sibling with NYT, then update the weight and P to the parent node.
      iii. If P is an internal node, update P, increase the weight by one, and adjust the nodes accordingly.
3. Repeat the above steps until the message is completed, and for each iteration update the pointer accordingly.

For the code generation, adaptive Huffman coding uses two types of codes. These codes are NYT and Fixed codes.

NYT codes are extracted from the trees. Fixed codes for a source containing m symbols $a_1, \ldots, a_m$ are generated using the following conditions:

1. For a given symbol, if the symbol number k lies between 0 and $2r$, then the symbol is encoded as the binary representation of $(k-1)$ in $(e+1)$ bits.

2. Else the letter is encoded as the binary representation of $(k-r-1)$ in e bits.

Here e and r are extracted from the following relations:

$$m = 2^e + r \qquad 0 \leq r \leq 2^e \tag{4.1}$$

Example 4.1. Consider the following string:

$$bbscwbsl$$

The values of e and r can be calculated from Equation 4.1 and found to be:

$$e = 5 \quad r = 10.$$

The adaptive code for this string is:

1. First symbol of the string is b for which $k = 2$. As initially the tree is empty, there is no NYT code as shown in Figure 4.1.

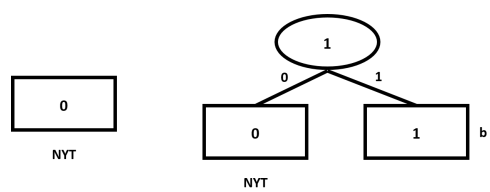

Figure 4.1: Symbol b

On the other hand for the fixed code $k < 2r$, therefore the code will consist of $e = 5$ bits and it will be the binary representation of $k - 1 = 1$. So the fixed code will be 00001 and the overall Huffman code for symbol b is given by:

$$\text{Huffman code for symbol b} = 00001$$

2. The second symbol of the string is also b and as it is already part of the tree, the code for this symbol by traversing the tree is as follows:

$$\text{Huffman code for symbol b} = 1.$$

The updated tree is given in Figure 4.2.

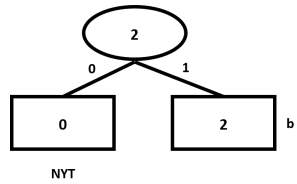

Figure 4.2: Symbol b

3. The next symbol of the string is s for which $k = 19$. The updated tree is presented in Figure 4.3.

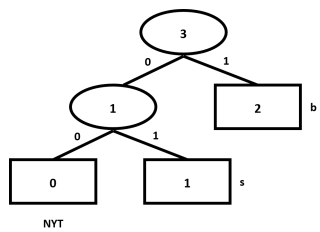

Figure 4.3: Symbol s

The NYT code is 0 and for fixed code $k < 2r$, therefore the code will consist of $e = 5$ bits and it will be the binary representation of $k - 1 = 18$. So the fixed code will be 10010 and the overall Huffman code for symbol s is given by:

$$\text{Huffman for Symbol b} = 010010$$

4. The next symbol is c for which $k = 3$. The updated tree is presented in Figure 4.4.

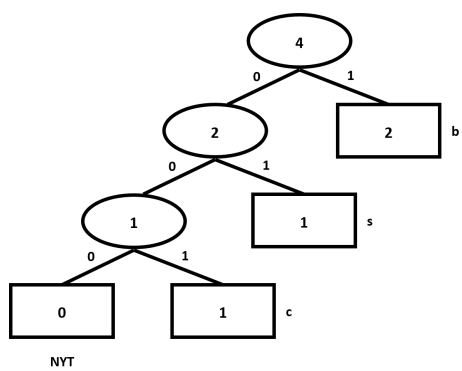

Figure 4.4: Symbol c

The NYT code is 000 and for fixed code $k < 2r$, therefore the code will be 00010 and the overall Huffman code for the symbol c is given by:

$$\text{Huffman code for symbol b} = 00000010$$

5. Now, w is the next symbol for which $k = 23$. The updated tree is presented in Figure 4.5.

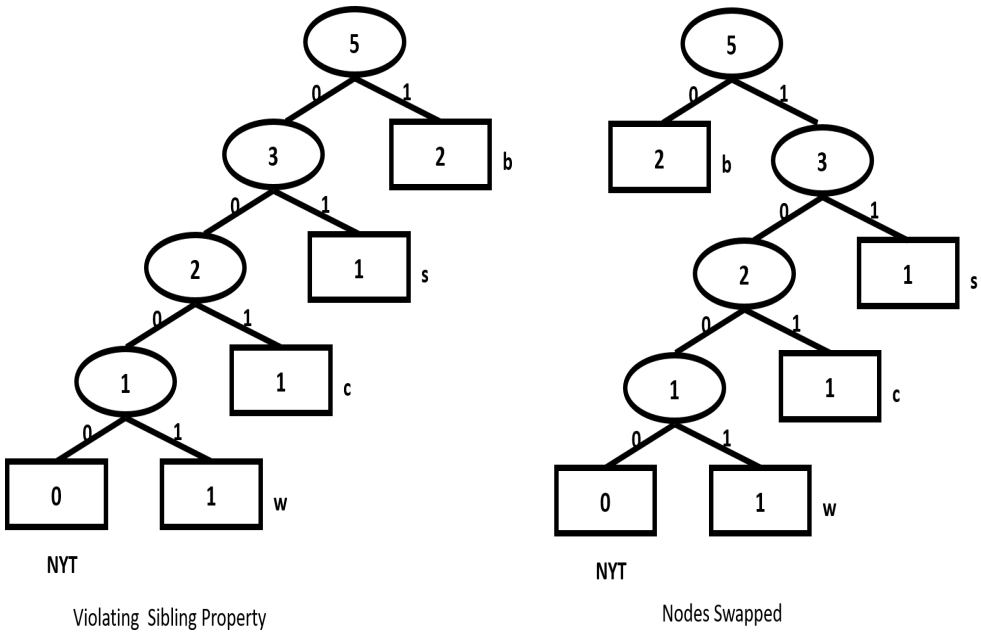

Figure 4.5: Symbol w

The NYT code is 000 and for fixed code $k > 2r$, therefore the code will consists of $e - 1 = 4$ bits and it will be the binary representation of $k - r - 1 = 12$. So the fixed code will be 1100 and the overall Huffman code for the symbol w is given by:

$$\text{Huffman code for symbol w} = 0001100$$

6. Next symbol of the string is also b and as it is already part of the tree, so the code for this symbol by traversing the tree is as follows:

$$\text{Huffman for symbol b} = 0 \tag{4.2}$$

The updated tree is given in Figure 4.6.

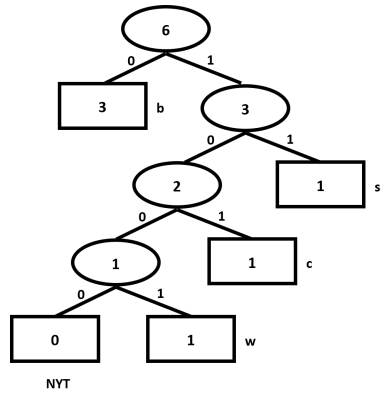

Figure 4.6: Symbol b

7. Next symbol of the string is s and as it is already part of the tree, the code for this symbol by traversing the tree is as follows:

$$\text{Huffman code for symbol s} = 10 \tag{4.3}$$

The updated tree is given in Figure 4.7.

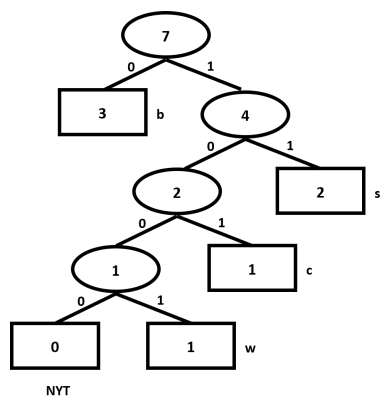

Figure 4.7: Symbol s

8. The last symbol is l for which $k = 12$. The updated tree is presented in Figure 4.8.

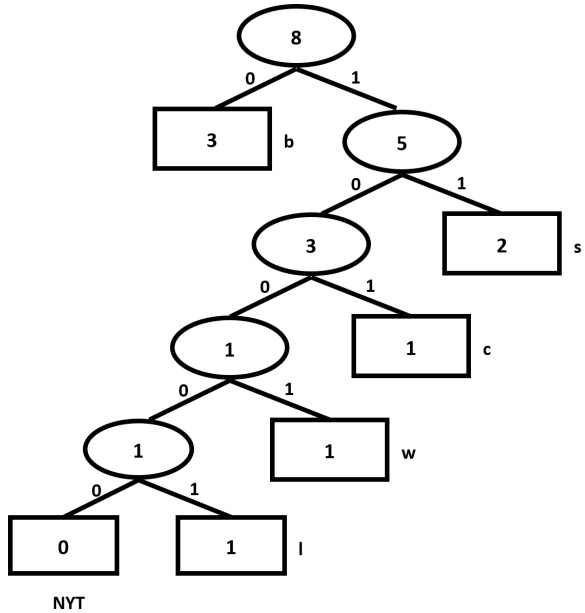

Figure 4.8: Symbol l

The NYT code is 1100 and for fixed code $k < 2r$, therefore the code will consists of $e = 5$ bits and it will be the binary representation of $k - 1 = 11$. So the fixed code will be 1011 and the overall Huffman code for symbol k is given by:

$$\text{Huffman code for symbol k} = 11000101$$

4.1.2 Steps for Adaptive Huffman Decoding

The decoding is similar to the encoding and it works by reading the encoded string. If the leaf node is NYT then read the next e bits and compare it with r.

1. If $e < r$, then convert the $e + 1$ bits to decimal value and add one to it.
2. If $e > r$, then convert the e bits to decimal value and add $r + 1$ to it.

4.2 Adaptive Arithmetic Coding

As discussed in the previous chapter, instead of encoding separate symbols the arithmetic coding codes the complete message with a single code. However, just like the standard Huffman coding, the arithmetic coding also needs complete message to encode it. For many applications, it is desirable to encode the data on the run-time.

To overcome this challenge, different types of adaptive arithmetic coding schemes have been introduced (see [1, 28, 29]). All these schemes update the probabilities of the symbols on the run-time and can be used for the complex models. Unlike the major differences between the Huffman and adaptive Huffman coding schemes, there is only a small difference between the static and adaptive arithmetic schemes.

4.2.1 Steps for Adaptive Arithmetic Coding

Most of the algorithm for the Adaptive Arithmetic Coding is the same as the static version of the scheme, and to avoid repetition, only the changes are discussed below:

1. For the case of adaptive arithmetic coding, as the message is encoded in real time, therefore, frequency table or the probability table is not used. To replace this table, all the symbols are assigned the same probability at the start.
2. As the length of the message is not known, the message length is not used and it is replaced by a big number.
3. As the frequency information and the message length is not required for the encoding and decoding, this scheme has low overhead.

The required input consists of a message to be encoded and the frequency table or probability of the symbols that is initialized to some constant at the beginning and updated as the message is being read. The main steps are as follows:

1. First step is to assign the probability of each symbols in the message and it is constantly updated. As for the case of adaptive arithmetic coding, the message is encoded in real time, therefore, frequency table is replaced and all the symbols are assigned the same probability at the start.

   For example, if the message to be encoded consists of five symbols α_1, α_2, α_3, α_4, α_5 and, as the message length is also unknown, a large number is used for the total number of symbols in the message. All the symbols will be assigned an equal probability, as given in the table below:

Letter	Probability
a_1	0.2
a_2	0.2
a_3	0.2
a_4	0.2
a_5	0.2

2. Just like static arithmetic coding, the adaptive arithmetic coding also uses cumulative probabilities of the symbols, which are represented on the line ranging from 0.0 to 1.0, where each symbol uses a sub-range. For a symbol α the sub range (C to E) is calculated using the following formula:

$$E = C + P(\alpha) \times R, \tag{4.4}$$

   where C is the cumulative probability, $P(\alpha)$ is the probability of symbol (α), and R is the range of the line. One thing to note that initially the line starts from 0.0 and ends at 1.0, thus $R = 1.0$.

   For the above example, the symbol α_1 will start from $C = 0$ and end at $E = 0 + 0.2(1) = 0.2$. Similarly, the table below presents the values for the remaining symbols:

Letter	Start	End
a_1	0	0.2
a_2	0.2	0.4
a_3	0.4	0.6
a_4	0.6	0.8
a_5	0.8	1.0

3. Starting from interval $[0, 1)$ with the partition given by the cumulative probabilities, the range of the interval will be restricted to the sub interval of the symbol being encoded. For example, if the message is $\alpha_2\alpha_4\alpha_3\alpha_5\alpha_1$, then the new range is equal to the one corresponding to α_2. Hence,

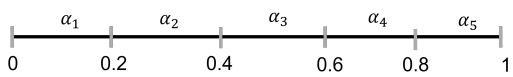

Figure 4.9: Adaptive Arithmetic coding for symbol α_2

4. The remaining steps are similar to the ones in the arithmetic coding, except for the need to update the probability at each step. The overall arithmetic encoding process for the message $\alpha_2\alpha_4\alpha_3\alpha_5\alpha_1$ is presented in Figure 4.10. So instead of separate codes for the symbols, a value between 0.3424 and 0.3427 is used.

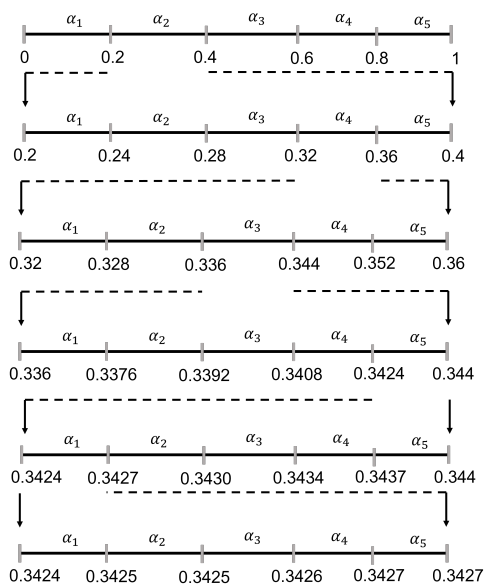

Figure 4.10: Adaptive Arithmetic encoding for symbol message $\alpha_2\alpha_4\alpha_3\alpha_5\alpha_1$

4.2.2 Steps for Adaptive Arithmetic Decoding

The decoding process consists of several steps as discussed below:

1. Assign equal probabilities to all the symbols that can be in the message. After that the decoding process is similar to the encoding one. Start with the decoding line as shown in Figure 4.11.

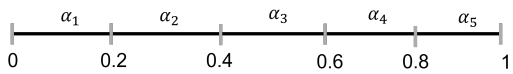

Figure 4.11: Arithmetic Decoding Step 1.

2. Search for the encoded value interval. In the example, the encoded value is 0.3425 and it lies between 0.2 and 0.4. So the first symbol should be α_2.
3. Restrict the sub-interval to $0.2 - 0.4$ and update the intervals just like the encoding.
4. Repeat the previous step until done..

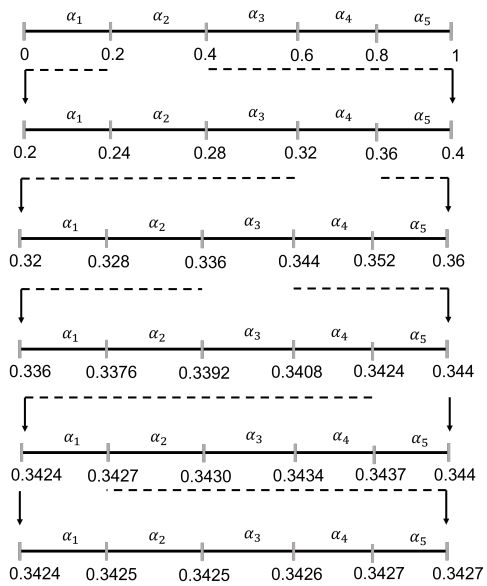

Figure 4.12: Adaptive Arithmetic Decoding with encoded message= 0.3425.

From the Figure, we deduce that the decoded message is $\alpha_2\alpha_4\alpha_3\alpha_5\alpha_1$.

Chapter 5

Experimental Results

In the previous chapters, the theories of Huffman coding, arithmetic coding, and their adaptive counter part versions were presented. In this chapter, we proceed with experiments on these algorithms using various simulation techniques in order to validate and compare their performances. Matlab built-in routines in conjunction with our implementation of additional functions are used to carry out the experiments. The chapter is divided into two sections. The first section presents the results for the coding scheme examples discussed in the previous chapters. The second section is devoted to the comparison between these techniques. In order to evaluate the compression efficiency of the different methods, we use the compression ratio which we recall is defined by the formula

$$C = \frac{\text{average codeword length}}{\text{average message length}}.$$

From the definition, it is clear that the smallest the value of C is the more efficient is the associated compression scheme.

5.1 Huffman Coding

In order to use the built-in Matlab routine *huffmanenco(I,H_{dic})* that takes as input a signal I and the Huffman codes described by the input code dictionary H_{dic}, we need to perform the following steps:

1. Read the input I
2. Find the unique vector of symbols $symbols = (s_1, \cdots, s_n)$ in the input
3. Calculate the number of times each symbol of the source alphabet occurs in the input
4. Calculate the probability vector $p = (p_1, \cdots, p_n)$ for each distinct symbol using the previous step
5. Using the probabilities and the distinct symbols, build the Huffman dictionary (H_{dic}) using the built-in Matlab function *huffmandict(symbols,p)*
6. Get the encoded sequence (H_{enc}) using the *huffmanenco* function
7. Convert the input to binary format I_b

8. Calculate the bits in the I_b and (H_{enc})
9. Calculate the compression ratio.

For the simulation, the parameters of Example 3.2 presented in Chapter 3 are used. The information about the symbols and their probabilities is reproduced in the table below:

Letter	Probability
a_2	0.4
a_4	0.2
a_1	0.15
a_3	0.15
a_5	0.1

The resulting Huffman codes are given in the table below:

Letter	Probability	Codeword
a_1	0.15	001
a_2	0.4	1
a_3	0.15	010
a_4	0.2	000
a_5	0.1	011

Note that the difference in codes obtained here and in Example 3.2 consists of the inversion in the bits and is due to the interpretation of the positions of the left and right children in a binary tree. For testing, we use the message $\alpha = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$, where $\alpha_i = a_i$. The binary code for the encoded message E_n is given in the equation below:

$$E_H = 0011010000011 \tag{5.1}$$

Decoding of the encoded binary stream is also performed and the resulting sequence is given below:

$$D_H = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$$

From the result of the encoded sequence, it can be seen that a total of ten bits is required to encode the message α using Huffman coding.

5.2 Arithmetic Coding

In order to use the built-in Matlab routine *arithenco(symbols,counts)* that takes as input a sequence of distinct symbols *symbols* and the variable *counts* (vector) that specifies the statistics of the source by listing the number of times each symbol of the source alphabet occurs in a test data set, we need to perform the following steps:

1. Read the input I
2. Find the unique vector of symbols $symbols = (s_1, \cdots, s_n)$ in the input
3. Calculate the variable *counts*, which is the number of times each symbol of the source alphabet occurs in the input

4. Using the previous steps, get the encoded sequence (A_{enc}) using *arithenco(symbols,counts)*
5. Convert the input to binary format I_b
6. Calculate the bits in the I_b and (A_{enc})
7. Calculate the compression ratio.

We use again the same source and the same message $\alpha = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$, with $\alpha_i = a_i$ to test the arithmetic method.

We obtain the following result in decimal:

Iter.	Start a_1	Start a_2	Start a_3	Start a_4	Start a_5	End a_5
0	0	0.15000	0.55000	0.70000	0.90000	1.00000
1	0	0.02250	0.08250	0.10500	0.13500	0.15000
2	0.02250	0.03150	0.05550	0.06450	0.07650	0.08250
3	0.05550	0.05685	0.06040	0.06180	0.06360	0.06450
4	0.06180	0.06207	0.06279	0.06306	0.06342	0.06360
5	0.06342	0.06342	0.06351	0.06351	0.06360	0.06360

The binary code for the encoded message E_n and the decoded message D_c from the encoded message are given in the equations below:

$$E_A = 000011111111101000000 \tag{5.2}$$
$$D_A = \alpha_1\alpha_2\alpha_3\alpha_4\alpha_5$$

From the result of the encoded sequence, we can see that to encode this message using arithmetic coding, a total of twenty one bits is required.

5.3 Adaptive Huffman Coding

The testing of the compression quality of the Adaptive Huffman coding, a Matlab program was written for that purpose. The detailed steps on which the program is based are as follows:

- Input: Text file, String, Symbols
- Output: Adaptive Huffman coded binary sequence

1. Read the first symbol and calculate the values of e and r as presented in the previous chapter
2. Update the NYT and weights
3. Read the next symbol and see if it is already available in the tree or it is a new symbol.

   For this a function *tree_gen* was developed which updates the tree as per the adaptive Huffman rules as discussed in the previous chapter

   (a) If it is a new character (symbol), it is added as the right child of NYT and tree is updated accordingly

   (b) If it already exists in the tree then its weight and order is adjusted.

   To update the weights a separate function *update_tree* was developed which updates the existing symbols weights as per the adaptive Huffman rules.

4. Use the developed functions to get the codes for specific symbols and complete tree
5. Convert the message to binary format I_b
6. Calculate the bits in the I_b and encoded sequence
7. Calculate the compression ratio.

For the testing, we used the following string: *Adaptive Huffman Testing*
The developed program is used and results of the first few symbols are presented in Figure 5.1.

```
op =
    '00010'
op =
  1×3 cell array
    {'00010'}    {'0'}    {'00110'}
op =
  1×5 cell array
    {'00010'}    {'0'}    {'00110'}    {'00'}    {'00101'}
op =
  1×7 cell array
    {'00010'}    {'0'}    {'00110'}    {'00'}    {'00101'}    {'100'}    {'01101'}
```

Figure 5.1: Text file used for Huffman compression.

In this case, there are 16 distinct letters as given below:

$$AHTadefgimnpstuvN$$

Corresponding codes are given in the table below:

Letter	Code
A	011
H	11110
T	100
a	11101
d	11110
e	111110
f	101
g	0010
i	00111001
m	110
n	00110
p	0011110
s	11100
t	0011101
u	11101
v	000

For this string, 79 bits are required for the adaptive Huffman coding. On the other hand, by applying the Huffman coding to the same string, the encoded string contained 98 bits of code. Furthermore, the statistical information also needs to be transmitted. So, the adaptive Huffman coding clearly outperforms the regular Huffman coding.

5.4 Adaptive Arithmetic Coding

The testing of the compression quality of the Adaptive arithmetic coding, a Matlab program was written for that purpose. The detailed steps on which the program is based are as follows:

- Input: Text file, String, Symbols
- Output: Adaptive Arithmetic coded binary sequence

1. Read the first symbol
2. Update the NYT and weights
3. Read the next symbol and see if it is already available in the tree or it is a new symbol.

   For this a function *tree_gen* was developed which updates the tree as per the adaptive Huffman rules as discussed in the previous chapter

   (a) If it is a new character (symbol), it is added as the right child of NYT and tree is updated accordingly

   (b) If it already exists in the tree then its weight and order is adjusted.

   To update the weights a separate function *update_tree* was developed which updates the existing symbols weights as per the adaptive Huffman rules.
4. Use the developed functions to get the codes for specific symbols and complete the tree
5. Convert the message to binary format I_b
6. Calculate the bits in the I_b and encoded sequence
7. Calculate the compression ratio.

For the testing, we use the following string: *Adaptive Arithmetic Testing*
The developed program is used for the encoding and decoding and the results of the first few symbols are presented in the figure below:

1	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	0	0	0	1	0	0	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	0	0	1	1	1	0	0
1	0	0	1	0	0	0	0
0	0	0	1	1	1	1	1

Figure 5.2: Adaptive Arithmetic compression.

The program requires more time to do the encoding. Furthermore, the size of the developed code is greater than the static version.

5.5 Comparison of the Coding Schemes

Huffman, Arithmetic, adaptive Huffman and adaptive Arithmetic coding schemes are tested and compared with each other. All the coding schemes are tested on the same data and the same machine to ensure a fair comparison. The following data sets are used:

1. Random Symbols
2. String
3. Text file

The coding schemes are applied to the above data sets and the following two parameters are calculated:

1. Compression ratio
2. Coding time

5.5.1 Random Symbols

For the first experiment, the encoding schemes are tested on a random message consisting of 100, 500 and 1000 symbols. For a fair comparison, the same sequence is used for the four schemes. The data used for the experiment is given in Table 5.1.

Data Type	Random Symbols	Random Symbols	Random Symbols
Number of Unique Symbols	300	300	300
Message Size	100	500	1000
Number of Bits	900	4500	9000

Table 5.1: Data for Random Symbols

Generated messages are encoded using the Huffman, Arithmetic, adaptive Huffman and adaptive Arithmetic coding schemes. The encoded messages and their corresponding compression ratios are presented in Table 5.2.

Message Size	100	500	1000
Number of Message Bits	900	4500	9000
Huffman Encoded Bits	639	3875	8025
Arithmetic Encoded Bits	611	3569	7512
Adaptive Huffman Encoded Bits	237	1405	2871
Adaptive Arithmetic Encoded Bits	220	1203	2510
Huffman Compression	0.71	0.86	0.89
Arithmetic Compression	0.68	0.79	0.83
Adaptive Huffman Compression	0.26	0.31	0.32
Adaptive Arithmetic Compression	0.24	0.27	0.28

Table 5.2: Data for Random Symbols

Figure 5.3 shows a more detailed comparison of the different schemes, where the number of symbols is presented on the x-axis and the corresponding compression ratio is given on the y-axis.

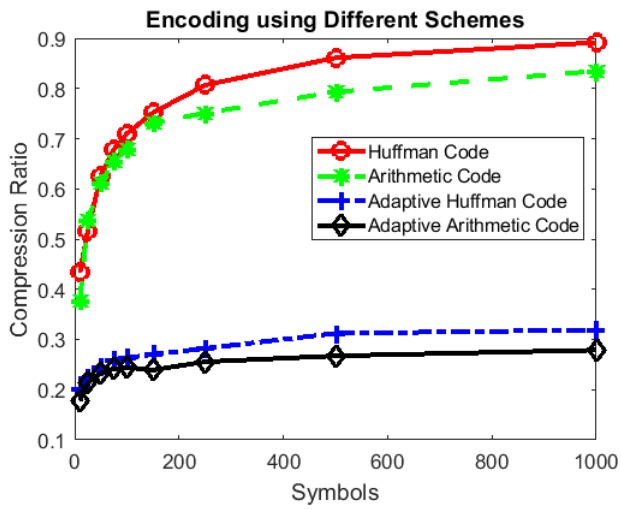

Figure 5.3: Random symbols compression using different schemes

From the figure, it can be seen that for smaller sizes Huffman coding and arithmetic coding exhibit the same performance, however, as the message size increases, the arithmetic coding works better than the Huffman coding. Furthermore, the adaptive versions of the Huffman and arithmetic codings are comparable for small sizes and the adaptive arithmetic coding is slightly better than the adaptive Huffman coding as the number of symbols increases. Overall, the adaptive versions of the coding schemes outperform the standard versions. As per the comparison, the adaptive arithmetic coding provides the best compression ratio out of these four tested schemes.

The running times for these compression algorithms are also computed and the results are presented in the table below.

Message Size	100	500	1000
Huffman Encoding	1.3	3.2	7.2
Arithmetic Encoding	4.5	10.6	16.1
Adaptive Huffman Encoding	2.7	5.5	12.3
Adaptive Arithmetic Encoding	5.7	12.2	18.6

Table 5.3: Run time comparison of the Encoded schemes for the Random Symbols

From the table, it can be deduced that the traditional Huffman coding is the fastest coding scheme, and the adaptive arithmetic coding is the slowest among the four tested schemes.

5.5.2 Random Strings

For the second experiment, the encoding schemes are tested on five different strings and to ensure a fair comparison, the same stings were used for the four schemes. The strings used for the experiment are given below:

1. abcdefghijklmnopqrstuvwxyz

2. scauchu32yr8f932hvn4288tug fi2h4uhvi3jgi3./qcjqwojvnviw @f9u9jmbkeo-:fq
3. This is a test string
4. This string will be encoded using the Huffman, Arithmetic, adaptive Huffman and adaptive Arithmetic coding schemes.
5. Compression ratio calculation.

The data used for the experiment is given in Table 5.4.

String Number	Unique Symbols	Message Length
1	26	26
2	32	71
3	11	21
4	26	115
5	16	30

Table 5.4: Data for Random Strings

Generated messages are encoded using Huffman, Arithmetic, adaptive Huffman and adaptive Arithmetic coding schemes. The encoded messages and their corresponding compression ratios are presented in Table 5.5.

String Number	1	2	3	4	5
Huffman Encoded Bits	124	340	68	500	116
Arithmetic Encoded Bits	129	346	74	504	123
Adaptive Huffman Encoded Bits	52	64	22	205	32
Adaptive Arithmetic Encoded Bits	49	61	24	198	30
Huffman Compression	0.68	0.68	0.46	0.62	0.55
Arithmetic Compression	0.71	0.69	0.50	0.63	0.59
Adaptive Compression	1	2	3	4	5
Adaptive Compression	1	2	3	4	5

Table 5.5: Data for Random Strings

Figure 5.4 shows a more detailed comparison of the different schemes, where the string numbers are presented on the x-axis and the compression ratios are given on the y-axis. From the figure, it can be deduced that adaptive versions of the Huffman and arithmetic codings are better than their standard versions. Also, for strings, adaptive Huffman and adaptive arithmetic codings both have the same compression performance. Figure 5.5 provides the computational time required for the encoding of the given strings using Huffman, Arithmetic, adaptive Huffman and adaptive Arithmetic coding schemes.

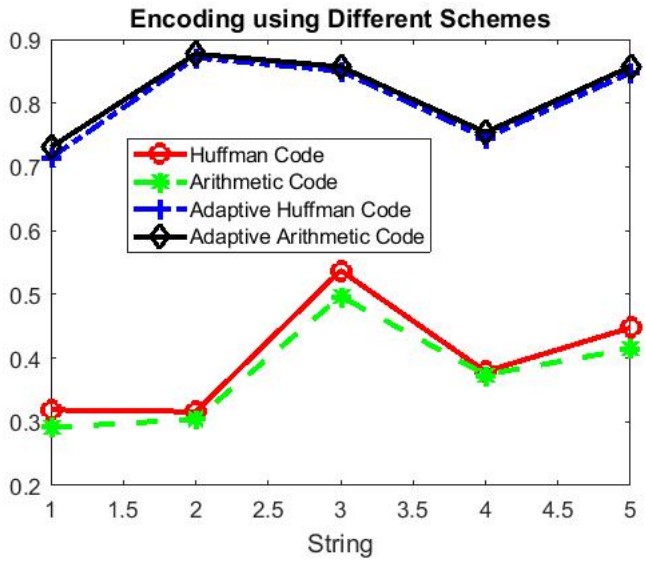

Figure 5.4: String Compression using different schemes.

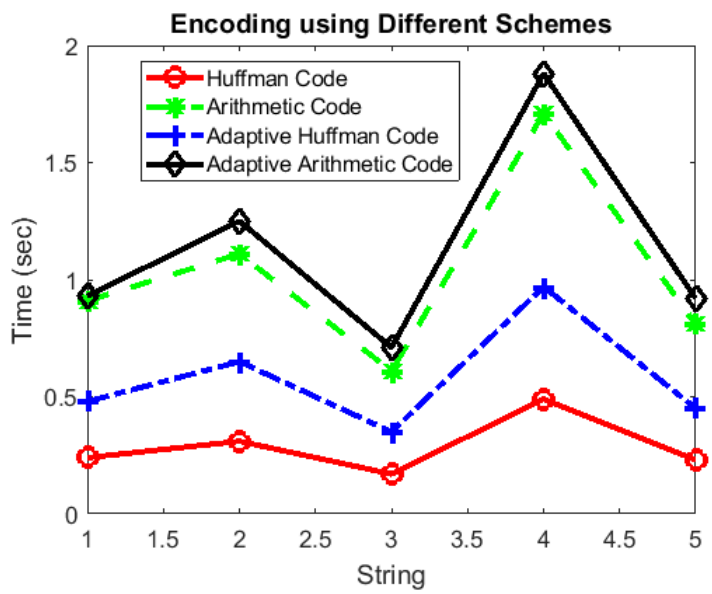

Figure 5.5: String Compression using different schemes

From Figure 5.5, it is evident that for random strings data, Huffman compression is the fastest encoding scheme. Furthermore, from Figure 5.4 and Figure 5.5, it can be deduced that for random

strings data compression, adaptive Huffman coding should be used, as it provides the best compression with acceptable computational time.

5.5.3 Text File Compression

For the last experiment, the four encoding schemes are applied for the compression of different text files. Details about the text files used for this experiment are provided in Table 5.6.

Number	Information	Size
1	Acknowledgment of the Thesis	399 B
2	Introduction of Chapter 1	527 B
3	Abstract of the Thesis	1.27 KB
4	Chapter 2	7.5 KB
5	Chapter 1	16.8 KB
6	Chapter 1-2	23.3 KB

Table 5.6: Data for Text Compression

These files are used to test the compression ratio of Huffman, Arithmetic, adaptive Huffman and adaptive Arithmetic coding schemes. The results of the encoding are presented in Figure 5.6, where the text file is presented on the x-axis and the corresponding saving is given on the y-axis.

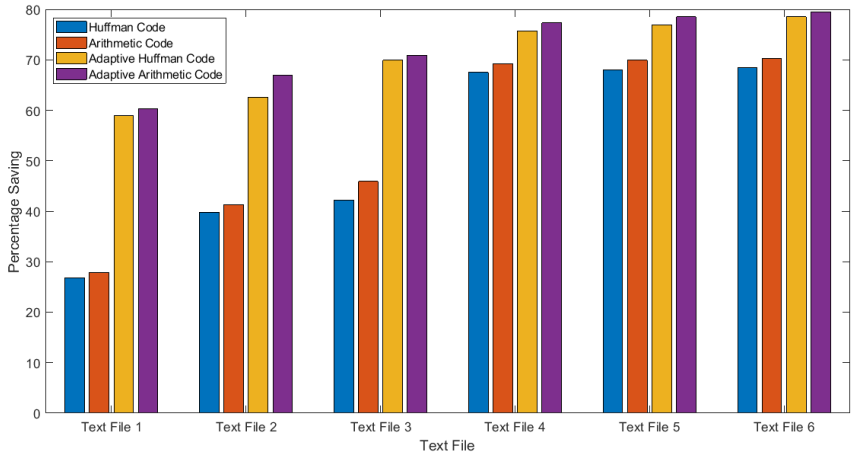

Figure 5.6: Text file Compression using different schemes

As discussed in the previous experiments, the adaptive arithmetic scheme provides a better compression ratio than Huffman, Arithmetic, and adaptive Huffman coding schemes. The rule of thumb is as the size of a text file increases, some symbols may occur more frequently. The direct effect is the increase of their corresponding probabilities which allows to achieve a smaller average codeword length and hence a smaller compression ratio. This is obviously equivalent to a higher percentage of saving.

Conclusion

Data compression is a subject of much significance and is of key importance in numerous modern applications. Strategies for efficient compression have been sought and studied for almost forty years.

In this thesis, we covered the theory of lossless data compression. We have given an outline of data compression strategies for general utility. Our objective is not to be exhaustive in covering the various methods of lossless compression. We rather focused on the two most known and widely used methods which are the Huffman and the Arithmetic coding methods and their adaptive counter part versions. These two methods exhibit different methodologies of coding and different strategies of implementation.

In Chapters 3 and 4, we have developed the details of the coding strategies for both methods and related them to the information theory concepts discussed in Chapter 1. The algorithms have been evaluated in terms of the amount of compression they provide as related to the entropy and also the algorithmic efficiency.

In Chapter 5 we carried out experiments on these algorithms using various methods aiming at validating and comparing their performances. In order to ensure a fair comparison, all schemes were tested on the same data. In addition to validating and confirming the projected performances for the individual methods, the comparison showed that the adaptive arithmetic coding provides the best compression out of these four tested schemes. The parameters we tested the algorithms on were the compression ratio and the computer coding time. However there is much left to explore about these algorithms and their future scope.

The main goal of this work is to assemble all the necessary concepts related to Huffman and Arithmetic coding in a single document. We have provided a concise algorithm with the complexity analysis of the Huffman coding method in Chapter 3. We have illustrated all the methods with simple examples. The good understanding of these methods can certainly help to develop increasingly effective ways to store and transmit information. For instance, one direction to explore is to seek approaches that maximize the effectiveness of arithmetic coding and adaptive arithmetic codings by efficiently partitioning the data. Another direction of future research is to focus on developing probabilistic models that are highly effective in modeling complex dependencies in data.

Bibliography

[1] Sayood K. *Introduction to Data Compression*. Morgan Kaufmann, fifth edition, 2017.

[2] Minnen D. et al. Joint autoregressive and hierarchical priors for learned image compression. *Advances in Neural Information Processing Systems*, 31:10771–10780, 2018.

[3] Tschannen M. et al. Deep generative models for distribution-preserving lossy compression. *Advances in Neural Information Processing Systems*, 31:5929–5940, 2018.

[4] M. Hosseini. A survey of data compression algorithms and their applications. *Appl. Adv. Algorithms*, DOI:10.13140/2.1.4360.9924:21–14, 2012.

[5] Uthayakumar J. et al. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University-Computer and Information Sciences*, 33:119–140, 2021.

[6] Vandenberg R. et al. Analyzing and improving integer discrete flows for lossless compression. arXiv: 2006.12459 [cs, stat], 2020.

[7] Ruan Y. et al. Improving lossless compression rates via monte carlo bits-back coding. arXiv: 2102.11086 [cs, math, stat], 2021.

[8] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. IRE 40, 9*, pages 1098–1101, 1952.

[9] Shannon C. A mathematical theory of communication. *Bell Sys. Tech. J.*, 27:623–656, 1948.

[10] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. John Wiley & Sons Inc., 1991.

[11] Borda M. *Fundamentals in Information Theory and Coding*. Springer-Verlag, 2011.

[12] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys (CSUR)*, 19 (3):261–296, 1987.

[13] R.J. McEliece. *The Theory of Information and Coding*. Addison-Wesley, 1977.

[14] Roy. Hoffman. *Data Compression in Digital Systems*. New York: Chapman & Hall, 1997.

[15] J. Abel and B. Teahan. Universal text preprocessing for data compression. *IEEE Transactions on Computers*, 54(5):497–507, 2005.

[16] Sz. Grabowski P. Skibinski and S. Deorowicz. Revisiting dictionary-based compression. *Software - Practice & Experience*, 35(15):1455–1476, 2005.

[17] M. Mahoney. Adaptive weighing of context models for lossless data compression. Technical Report CS-2005-16, Florida Institute of Technology, 2005.

[18] R. C. Gonzalez and R. E. Woods. *Digital Image Processing*. Pearson, 2008.

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

[20] J. Rissanen. Complexity of strings in the class of markov sources. *IEEE Transactions on Information Theory*, IT-32:526–532, 1986.

[21] J. Rissanen M. J. Weinberger and M. Feder. A universal finite memory source. *IEEE Transactions on Information Theory*, IT-41:643–652, 1995.

[22] Fano R. M. *Transmission of Information*. MIT Press, 1961.

[23] R.G. Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, IT-24:668–674, 1978.

[24] N. Abramson. *Information Theory and Coding*. McGraw-Hill, NY, 1963.

[25] N. Faller. An adaptive system for data compression. In *7th Asilomar Conference on Circuits, Systems and Computers*, pages 593–597, 1973.

[26] D. E. Knuth. Dynamic huffman coding. *J. Algorithms*, 6 (2):163–180, 1985.

[27] J. S. Vitter. Design and analysis of dynamic huffman codes. *J. ACM*, 34 (4):825–845, 1987.

[28] Bentley J.L. et al. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986.

[29] Witten I. H. et al. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–541, 1987.