Department of Electrical and Computer Systems Engineering
Monash University

Information and Networks, ECE3141

Lab 2: Entropy Coding

Authors: Dr. Mike Biggar
Dr. Gayathri Kongara
(updated 16 March 2022)

1. Introduction

We have hseen in lectures that "Information" is something that can be quantitatively measured, and it is determined by the "surprise" or "uncertainty" of an event[1]. So, a very likely event (e.g. warm temperatures experienced in summer) conveys little information when it occurs, whereas a less likely event (snow falling in summer) gives us much more information.

We further saw that the average information rate, or "Entropy", of a sequence of events tells us the absolute minimum number of transmitted or stored bits needed to communicate that sequence of events (on average).

A third key concept introduced in lectures was the difference between "binary digits" (the 1s and 0s used to communicate information) and information "bits" (the actual information carried). If we have an inefficient communication system, we might send a lot of binary digits without conveying that much information. However, the entropy measure is important because it tells us the lower limit on the number of binary digits needed to communicate the sequence of events. We cannot go below this without losing information.

The mathematical calculation of entropy (mean information content of a set of discrete, independent data) tells us the minimum average number of bits needed to communicate that data without information loss. (See box below.) The entropy is calculated from the probabilities of the individual possible event outcomes, which collectively make up the probability mass function (pmf) *p(X):*

$$H = \sum_i p(x_i) I(x_i) = -\sum_i p(x_i) log_2(p(x_i))$$

where $I(x_i) = -\log_2(p(x_i))$ is the information associated with the event $x = x_i$. Entropy is maximised when all the outcomes are equally likely; that is, we get the same amount of information when we learn each outcome. So, for example, if we have 8 possible outcomes and they are all equally likely ($p(x_i) = \frac{1}{8}\ for\ all\ i$), substitution in the above expressions will show that $I(x_i) = 3\ for\ all\ i,$ and *H=3* bits. In this case, we would allocate one of the eight possible 3-bit words to identify each of the 8 possible outcomes and knowing that H=3 tells us that no other code word representation can do any better; we require at least 3 bits per code word.

[1] An "event" could be many things but in telecommunications it is commonly a symbol or message to be transmitted or stored.

However, if the outcomes are not equally likely (for instance, Figure 1 shows the pmf for pixel intensities in an example image), then we learn more from a low-probability event (because it is less expected) and we learn less (that is, we gain less information) from a more probable event. In this case, we can represent our data more efficiently by using fewer bits (binary digits) to identify the more likely outcome, and more bits to identify the less likely outcome[2].

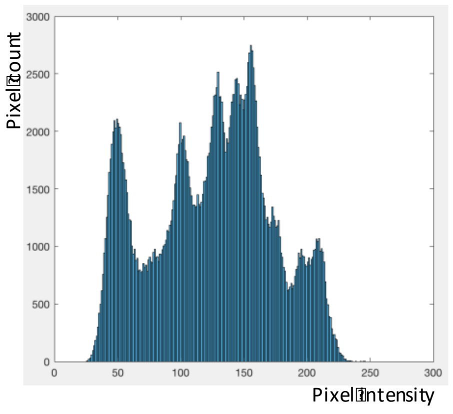


Figure 1. An example of a common data source (pixel intensities in a "typical" image), demonstrating different frequencies of occurrence (i.e. different probabilities) for different intensities.

"Information loss"?

It is important to understand that the "information loss" referred to above has nothing to do with bit errors or problems with transmission. It is concerned only with representing the original data. If the entropy calculation says there is an average H bits/event of information, then that's how many bits we need to send (on average) to describe a series of such events; any less and we cannot recover the original data.

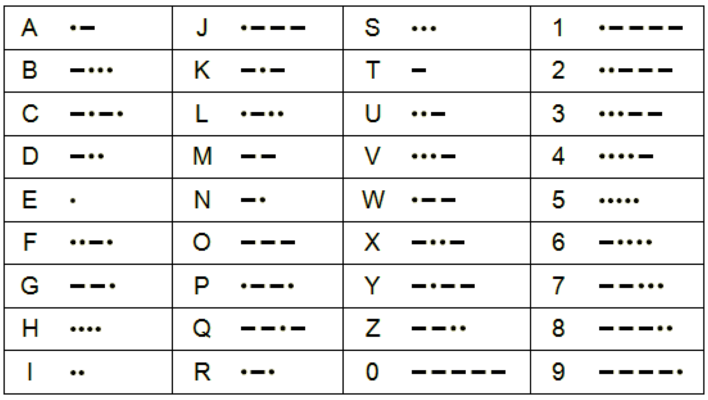
For example, suppose we are measuring temperature 5 times per second. Maybe each measurement comes from the digital thermometer as 1 byte (8 bits). We can (obviously) communicate the temperature data if we send 5 x 8 = 40 bits per second, but perhaps we don't need that much. After accumulating lots of temperature measurements, we can estimate the pmf and that allows us to calculate the entropy; let's say it is 2.3 bits per temperature data point. (It's an average, so it doesn't have to be an integer). Without knowing <u>how</u> we could represent the data (this will be explored later in this lab), we now know that we MUST transmit AT LEAST 5 x 2.3 = 11.5 bits per second (on average) if we're to be able to recover the original temperatures at a receiver somewhere. If we only have 10 bits per second available, it is <u>impossible</u> (no matter how we represent the data) to recover the original temperatures. There will be information loss even if those 10 bits per second are communicated with absolute accuracy (no errors).

Note, however, that the entropy calculation assumes that each x_i is independent. In our temperature example (and in many other cases), they may not be. If we know the temperature at one time instant, we might be able to make a good guess about what the next one will be. In this case, we might decide to transmit temperature differences (that is, the <u>change</u> in temperature from 200 ms ago). In this case, we might expect that the pmf of this differential data is less uniform than that of the independent temperatures, will lead to a still lower entropy, and therefore the data could be represented with fewer average bits.

[2] Don't worry if it is not intuitively obvious that this is necessarily true. The first exercise below should convince you of it.

The important point to understand is that, however we model the data, the uncertainty about each new value means that there is an absolute lower limit to the number of bits needed to remove that uncertainty at the receiver.

Again as we discussed in lectures, a common example of a variable length code is Morse code from the early days of telegraphy (see Figure 2). Note that the more commonly occurring letters (e.g. e, t) have been assigned shorter code words than less commonly occurring letters (like q or j).



A	•–	J	•–––	S	•••	1	•––––
B	–•••	K	–•–	T	–	2	••–––
C	–•–•	L	•–••	U	••–	3	•••––
D	–••	M	––	V	•••–	4	••••–
E	•	N	–•	W	•––	5	•••••
F	••–•	O	–––	X	–••–	6	–••••
G	––•	P	•––•	Y	–•––	7	––•••
H	••••	Q	––•–	Z	––••	8	–––••
I	••	R	•–•	0	–––––	9	––––•

Figure 2. Table of Morse Code (from http://www.bbc.co.uk/schools/gcsebitesize/science/ocr_gateway/home_energy/light_and_lasersrev1.shtml).

If we have a variable length code (like Morse code), an important parameter is the average number of bits/symbol. Suppose we have a set S of i symbols, each symbol s_i is represented by a binary code word of length L_i bits (i.e. L_i is not the actual code word, but the number of bits in the code word that represents s_i) and occurs with probability p_i. Write down an expression for the average number of bits/symbol in the box below.

Average number of bits/symbol:

$\sum pi * Li$, where pi is the probability of the symbol occuring and Li is the number of bits in the code word that represents si.

The expression you've just written gives us the actual average number of bits/symbol, but it may or may not be equal to the entropy. It cannot be less than the entropy but it could be more, depending on how efficient the code is.

While the entropy formula tells us the minimum average number of bits needed, it doesn't tell us how to construct a set of code words that will achieve it. However, several suitable codes were developed in the second half of the 20th century, the most important of which are[3]:

- **Huffman Coding.** Given a set of (assumed independent) symbols of known probability to be represented, this technique will generate variable bit length codes (one code per symbol) that most closely match the average word length given by the Entropy, within the limitation that we use an integer number of bits for each code word.

[3] These are not the only variable length coding methods, and there are also several variations to each of them.

- **LZW (Lempel-Ziv-Welch) codes**. By building up a library of sequences of symbols at both encoder and decoder, the LZW coder allocates code words to a variable number of symbols.
- **Arithmetic Coding.** By interpreting a group of symbols as defining a segment on a number line, and then identifying that segment with an appropriate number of bits, arithmetic coding allocates a variable number of bits to a variable number of symbols. The advantage of arithmetic coding is that it can take us arbitrarily close to the entropy limit if the probabilities of the symbols are known and the symbol sequence is stationary (i.e. the statistics do not change with time).

These are examples of code word assignment schemes collectively known as "Entropy Coders" (to finally explain the title of this laboratory exercise!), or "Variable Length Codes" (VLCs). They are by no means the only methods available to us to reduce the bit rate we use to transmit or store digital data. Rather, they are tools that we can use to exploit the statistics of a set of "outcomes" generated by some other process, where those outcomes could represent anything from spreadsheet entries to text characters to speech sample magnitudes to colour descriptions of video. There are numerous other compression techniques that exploit, for instance, known characteristics of the source of the data (e.g. knowing that it is a speech signal) or how it will be used (e.g. knowing what components of an image our eyes are most sensitive to). All of these other compression methods, even if they give us much greater savings in bit rate[4] than entropy coding alone, will generate sets of symbols for transmission or storage, and they can all take advantage of entropy coding. In short, entropy coding is a component of very many communication and storage systems, even if it might not always be the main source of bit rate savings.

In this laboratory class, you will carry out the following as you investigate the representation of data using variable length codes:

- Calculate entropy for simple probability distributions
- Demonstrate that variable length codes can provide data rate advantages for non-uniform probability distributions
- Use the Huffman coding tools in Matlab to generate code words, and both encode and decode some simple data.
- Use the same tools to encode ASCII text data
- Investigate issues of code book generation at encoder and decoder, and whether the code book itself needs to be communicated

2. Pre-lab

Prior to the laboratory class, you must read through this entire laboratory description and complete the pre-lab online quiz. You may also be able to complete answers to some of the questions which are based on theory (i.e. that don't depend on experimental results).

You should also consider each exercise given, and think about the Matlab processing you need to do (that is, how will the algorithm work? What instructions will you need? Do you know how to use them?). In particular, you should familiarise yourself with the Huffman Coding functions and the use of cell arrays, as discussed in Section 4(b) below.

[4] These "much greater savings" are only really achievable if we can tolerate some loss, or distortion, of our data (i.e. the coder is "lossy"). Entropy coding is always lossless, and can achieve much less compression by itself, though it may be used in conjunction with lossy coding tools. Lossy coding applies when we are dealing with digitised versions of analogue (e.g. image or sound) data in which some limited signal distortion may not be noticeable or can be tolerated. You can learn much more about these different compression methods (such as JPEG, MPEG. MP3, etc.) in *ECE4146 Multimedia Technologies*.

3. Entropy and variable length coding

The following table lists a set of 16 symbols that could be used in a message we want to transmit (we're labelling them "a" to "p" but this, of course, is arbitrary), along with three different probability distributions and three different sets of code words we might use to represent the 16 quantities. For now, assume that any transmitted bits are received reliably by a decoder, so we do not have to worry about the effect of transmission bit errors[5].

Symbol s	$P_A(s)$	$P_B(s)$	$P_C(s)$	Code set 1	Code set 2	Code set 3	Huffman code based on P_B
a	0.0625	0.2	0.02	0000	001	101	[1,1]
b	0.0625	0.08	0.08	0001	0000	0000	[0,1,0,0]
c	0.0625	0.08	0.08	0010	0001	0001	[0,0,0,1]
d	0.0625	0.08	0.08	0011	0100	0100	[0,0,0,0]
e	0.0625	0.08	0.08	0100	0101	0101	[0,0,1,1]
f	0.0625	0.08	0.08	0101	0110	0110	[0,0,1,0]
g	0.0625	0.06	0.06	0110	0111	0111	[1,0,1,0]
h	0.0625	0.06	0.06	0111	1000	1000	[0,1,1,1]
i	0.0625	0.06	0.06	1000	1001	1001	[0,1,1,0]
j	0.0625	0.06	0.06	1001	1010	1010	[1,0,0,1]
k	0.0625	0.06	0.06	1010	1011	1011	[1,0,0,0]
l	0.0625	0.06	0.06	1011	1100	1100	[0,1,0,1]
m	0.0625	0.01	0.05	1100	1101	1101	[1,0,1,1,0,1]
n	0.0625	0.01	0.05	1101	1110	1110	[1,0,1,1,0,0]
o	0.0625	0.01	0.06	1110	11110	11110	[1,0,1,1,1,1]
p	0.0625	0.01	0.06	1111	11111	11111	[1,0,1,1,1,0]

Table 1. Table of alternative symbol probabilities and code words. You will fill in the last column when you complete Part 4a below.

Complete the following table of calculations and conclusions based upon the probabilities and code words above. (Hint: It's a little tedious but, if you like, you could do this by hand using a calculator, without using *Matlab*. However, you will need the probability vector in *Matlab* for the next step anyway, so it might be just as easy to use *Matlab* from the start.)

What is the entropy of the source if the probabilities are those described by P_A?	4 bits/symbol

[5] This may seem an unrealistic assumption, but in any practical system a source coding process (i.e. coding according to the properties of the data source) such as entropy coding will be followed by a channel coding process (i.e. coding according to the properties of the channel). The latter will add error protection (e.g. error correction bits, or an acknowledgement/retransmission protocol) to ensure that the probability of errors getting through to the source decoder will be very small. Management of such transmission errors is, of course, very important, but it's out of scope for this lab. We will explore it in the next one.

What is the average number of bits/symbol if the probabilities are described by P_A and we represent the symbols with Code Set 1?	4 bits/symbol
What do you conclude from your answers to the above two questions? Since the entropy is equal to the number of bits/symbols (all outcomes are equiprobable), we will achieve the maximum entropy. This means that Code Set 1 is the most efficient code set as it has 4 bits on every symbol.	
What is the entropy of the source if the probabilities are those described by P_B?	3.65 bits/ symbol
What is the average number of bits/symbol if the probabilities are described by P_B and we represent the symbols with Code Set 1?	4 bits/symbol
What do you conclude from your answers to the above two questions? There is some form of inefficiency in the coding compared with the coding for part (a) as the average number of bits/symbols is not equal to the entropy of the source (outcomes are not equally probable). Since the entropy of the source is 3.65 and the average number of bits/symbols is 4, this means that for every 4-bit transmitted, there will only be 3.65 bits of information.	
What is the average number of bits/symbol if the probabilities are described by P_B and we represent the symbols with Code Set 2?	3.82 bits/symbol
What do you conclude from your answer to the above question? This form of coding is more efficient compared with Pb with Code Set 1 as the average number of bits per symbol is closer to the entropy. This could be due to the change in the length of the code which will affect the bits/symbol.	
What is the average number of bits/symbol if the probabilities are described by P_C and we represent the symbols with Code Set 2?	4.1 bits/symbol
What do you conclude from your answer to the above question? Based on the visual observation made, the lower probability is assigned to words that are shorter in length while the larger probability is assigned to symbols longer in length.	
Code set 3 has the same code word lengths as Code set 2. The only difference is the first code word. What problem do you foresee in using Code set 3, that we would not have with Code set 2? Could Code set 3 be used in a practical system?	

The code word will be the prefix of another code word. In Code Set 3, “1010” has a prefix of “101”. It may not be suitable to be used in a practical system. Code Set 3 is also not uniquely decodable. The decoder may mistaken the valid code word.

4. Huffman coding

Your calculations have shown that, by appropriate allocation of variable length codes to represent symbols, we can obtain an advantage compared with use of fixed length code words. However, we don’t know if the code word sets we have in the table are the best that we can do. Perhaps there’s another set of variable length codes that can get us even closer to the limit defined by the entropy. Within the constraint of allocating an integer number of bits to represent any input symbol, and on the assumption that the symbols are independent, Huffman Coding will generate optimum code words; that is, there are no other variable length code words (that can be allocated one word per input symbol) which will get us closer to the entropy limit.

You have seen in lectures how to generate Huffman Codes, but there is a *Matlab* function that will generate them for you.

a) Read the Matlab documentation on the functions *huffmandict*, *Huffmanenco* and *huffmandeco*. Create one vector containing the symbols a..p, and another with the probabilities P_B from the earlier table. (Hint: When using character symbols, the symbol vector has to be a cell array.) Then use *huffmandict* to create the dictionary of code words. <u>Write the code words in the last column of the earlier table</u>. Here is some example code that may help, but you should not just copy and use it blindly; you need to understand what it is doing!

```
PB=[0.2 .08 .08 .08 .08 .08 .06 .06 .06 .06 .06 .06 .01 .01 .01 .01]
symbols=['a';'b';'c';'d';'e';'f';'g';'h';'i';'j';'k';'l';'m';'n';'o';'p'];
c_symbols=cellstr(symbols);
dict=huffmandict(c_symbols,PB);
dict{:,:}  % Display contents of dict array
```

Using the Huffman code words that you have now written in the earlier table, what is the mean number of bits/symbol if the symbols follow the P_B probabilities and we encode them using the Huffman code words?

Approximately 3.68 bits/symbol

What does the Huffman coder do if all the symbols have the same probability (i.e. as with P_A)?

If all of the symbols have the same probability as with Pa, each of the symbol will be assigned 4 bits code by the Huffman coder as the 16 symbols will be prefix free.

You might be thinking that the last question has been a bit contrived because we use 16 symbols, conveniently matched to a 4 bit code word. What happens if we don't have such a match? Suppose we had, say, 20 symbols that occur with equal probability (0.05). Before running the Huffman coder, ask yourself how you would allocate code words to represent such a set of symbols; would you just use 5 bits to identify each symbol, even though we don't have 32 symbols? What does the Huffman coder do when you present it with this problem?

I will not assign 5 bits to identify each symbol as it may cause a loss of information. I would however assign the different length of code word in random as the probability of each symbol are equal.

Now that we have an understanding of how Huffman coding works, let's move on from the artificial examples above and use it to compress some real data, in this case ASCII text. You will appreciate by now that we must have the probabilities of occurrence of each character if we are to generate a Huffman code. We will discuss more in the next section about what statistics we should use to generate the code table, but one way to obtain the probabilities is to analyse a lot of text (in the appropriate language) and count the frequencies with which each character occurs[6], then hope that the statistics are similar to the particular text we want to transmit. One such source was used to obtain the figures in Table 2. As an aside, compare some of the highest and lowest letter probabilities with the Morse code length in Figure 2, to confirm that they make sense. We need these characters and their probabilities[7] in appropriate arrays to allow us to generate Huffman codes in *Matlab*. To save you time importing and getting formats right, this data is available for you in the correct array formats, in the file "HuffmanText.mat". You will need to import this file into your Matlab data space, using a command such as `importdata()` or `matfile()`. You can use your own approach and initiative to do this but, if you get stuck, the following example code may help:

```
>> HuffmanObject = importdata('HuffmanText.mat');
>> symbols = HuffmanObject.AsciiChar
>> symbols_probs = HuffmanObject.AsciiProb
>> [dict2,Av_lenA] = huffmandict(symbols,symbols_probs) ;
```

Char	Prob	Char	Prob	Char	Prob	Char	Prob
		7	1.04E-03	O	1.85E-03	g	1.57E-02
(SPACE)	1.73E-01	8	1.06E-03	P	2.63E-03	h	2.76E-02
!	7.20E-05	9	1.03E-03	Q	3.18E-04	i	4.93E-02

[6] It's not even straightforward what this means! Some people have done character counts on dictionary words, but not all words in the dictionary are equally likely to be used. Others have performed such counts on, for example, the collected works of Shakespeare, but is this representative of the text you might be encoding? The probability data we are using here comes from someone who has analysed a year's worth of PDA memos ("PDA"?! Yes, it's old; from 2002.). This may also not be representative, but at least this data includes punctuation marks and capital letters, which are often omitted in these types of analyses. If you find a source of such statistics that you think is better, let us know!

[7] Note that the TAB character is missing from this data set. If you try below to process some text with a TAB in it, it will cause an error.

"	2.46E-03	:	4.38E-03	R	2.54E-03	j	8.73E-04
#	1.80E-04	;	1.22E-03	S	4.03E-03	k	6.80E-03
$	5.65E-04	<	1.23E-03	T	3.34E-03	l	3.20E-02
%	1.61E-04	=	2.29E-04	U	8.19E-04	m	1.65E-02
&	2.27E-04	>	1.25E-03	V	8.97E-04	n	5.00E-02
'	2.46E-03	?	1.48E-03	W	2.54E-03	o	5.81E-02
(	2.19E-03	@	7.33E-05	X	3.45E-04	p	1.56E-02
)	2.25E-03	A	3.15E-03	Y	3.06E-04	q	7.51E-04
*	6.32E-04	B	2.18E-03	Z	7.62E-05	r	4.29E-02
+	2.16E-04	C	3.93E-03	[	8.68E-05	s	4.40E-02
,	7.43E-03	D	3.17E-03	\	1.57E-05	t	6.41E-02
-	1.38E-02	E	2.69E-03	]	8.89E-05	u	2.11E-02
.	1.52E-02	F	1.43E-03	^	3.39E-06	v	8.52E-03
/	1.56E-03	G	1.89E-03	_	1.17E-03	w	1.31E-02
0	5.55E-03	H	2.34E-03	`	8.89E-06	x	1.96E-03
1	4.62E-03	I	3.23E-03	a	5.22E-02	y	1.14E-02
2	3.34E-03	J	1.74E-03	b	1.03E-02	z	6.00E-04
3	1.86E-03	K	6.92E-04	c	2.13E-02	{	2.63E-05
4	1.36E-03	L	1.90E-03	d	2.52E-02	|	6.78E-06
5	1.67E-03	M	3.55E-03	e	8.63E-02	}	2.58E-05
6	1.16E-03	N	2.10E-03	f	1.38E-02	~	3.39E-06

Table 2. An example set of measured probabilities of occurrence for letters and symbols in English.

b) Use the *huffmandict* function with the provided arrays to generate code words to represent each of the characters in the table according to the associated probabilities.

What is the longest code word generated? What is the shortest? Are these assigned to the expected characters?

The longest code word generate is 1111 0111 1000 0100 11 and 1111 0111 1000 0100 10 which are both 18 bits long. The shortest code word is 000 which is 3 bits long. Yes, it is very likely that the code word is assigned to the expected characters. Characters such as ^ and ~ are rarely used which explains why a long codeword is assigned to these characters while the "space" character has the shortest codeword since it is more frequently used compared with other characters.

c) So, now that we have a dictionary of variable length code words that should represent English text efficiently, let's try it out. Copy (or write) any random example of English text (e.g. from an email or some part of this document) into a text array in Matlab. Then encode it using `huffmanenco` and decode using `huffmandeco`. The

Matlab documentation gives guidance to calculate and compare the number of bits needed before and after Huffman encoding. Try a few examples.

Note that, since Matlab uses the single apostrophe character to delimit text strings, a piece of text that contains an apostrophe will cause it to terminate the string, and then the subsequent text will cause an error. You need to “escape” the apostrophe by inserting an extra apostrophe before each apostrophe in the string (so that, wherever the original text contained a ‘, it now appears as ‘‘) then terminate the string with a single apostrophe in the normal way.

By what percentage is the number of bits to represent a “typical” text string reduced using Huffman coding?

We would obtain 399 bits for the text string and 273 bits based on our encoded text string. 273/399 * 100 = 68.42% will be the total number of bits compressed. So, the percentage of the number of bits reduced would be 100 – 68.42 which equates to 31.58%.
The percentage of the number of bits reduced using Huffman coding is 31.58%.

5. Code word dictionaries

Hopefully you have realised an important requirement for a Huffman encoder and decoder to work together is that the code word dictionary, or “lookup table”, must be available to both encoder and decoder. Of course, the two must also be identical. There are two obvious approaches to this problem.

One approach would be to rely on globally agreed statistics (effectively what we have done in the previous exercise) and use the same code word dictionary for many communication sessions, either by communicating the code word dictionary once or have the encoder and decoder derive it based on the same probability data[8]. What would be the advantages and disadvantages of this approach?

Advantages:

- Potentially save cost and time as the same code word can be used time and time again. There is no need to create a new code word.
- There would also be a more coherent and systematic reference when we are assigning the bits
- Standardization of encoding and decoding of information will create convenience for everyone.

Disadvantages:

- Potentially waste memories if the globally agreed statistics is not accurate.

[8] This latter approach could be a risky one, because the two dictionary generators would have to run exactly the same algorithm. Huffman codes are not unique (that is, there can be multiple Huffman code word sets that give the same average bits/symbol), so we need to be confident that encoder and decoder generate the same one! If you’re unclear why Huffman codes are not unique, discuss with a demonstrator.

- The existence of risk that the dictionary generator may generate different code words.

The second approach is to analyse the message (or file, or image, or whatever) we want to send, obtain the probability statistics for our symbol set, generate the optimum code word set for this data set, and send the dictionary along with the message. What would be the advantages and disadvantages of this approach?

Advantages:

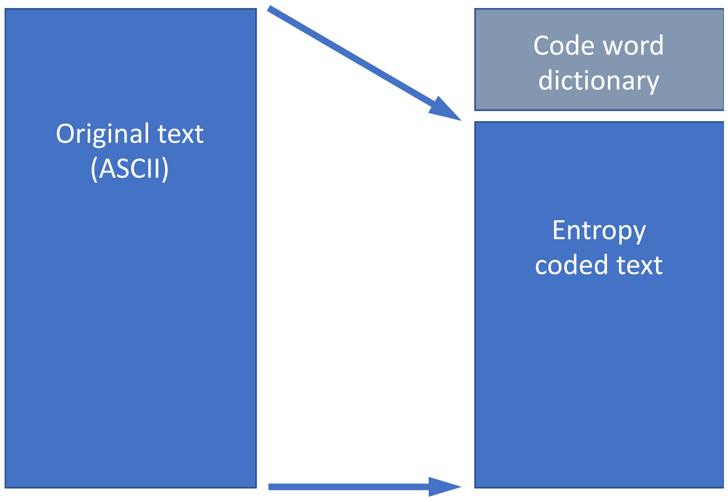
- A more efficient code word can be generated since we are using our own data set to determine which characters are more common than the others which may allow us to use a lower number of bits.

Disadvantages:

- There could be wasted resources in trying to create a new dictionary every time.

There is no single answer as to which of the above approaches is best. It really depends on the circumstances. Some video coding standards actually define, inside the standards document, the bit patterns of variable length codes to be used to represent certain data (based on statistics obtained during experiments carried out as the standard was developed). JPEG uses the alternative approach when coding certain image data[9]; the Huffman coding dictionary is embedded in every JPEG image file and can be chosen to best suit each one.

Clearly one important consideration in our choice between the above two approaches is whether we have the time and processing cycles available to analyse data statistics before defining variable length code words. In the case of video calling, we probably don't, since we need to send video description data as soon as possible to minimise end-to-end delay. Another is the size of the data file, as shown in figure 3 and the following example.



[9] Not pixel intensities directly, but other image descriptors. JPEG is based on use of a 2-dimensional transform called the Discrete Cosine Transform (similar to the Fourier transform), and it is the quantised transform coefficients that are Huffman encoded.

Figure 3. We compress our data with an Entropy Coder (such as Huffman Coder) but, if we have to add a code word look up table to decode it, have we saved anything? (The size of the blocks is supposed to indicate the amount of data required.)

Try to estimate how many bits it would take to represent the Huffman code word dictionary that was generated from the ASCII character statistics. Note: there is no single, correct answer to this question. Just think about what information a decoder would have to have to be able to receive the variable length codes and turn them back into the correct ASCII text, then come up with a method (it doesn't have to be super-efficient) to represent that information. Then calculate how many bits would be required.

Number of bits to represent word dictionary can be calculated by having the total number of symbols, the average number of code length, the length of each Huffman code and the number of bits needed to represent the symbols.

Based on the given text file, the total number of symbols that we have is 95. Then, we can get the length of each Huffman code which is found to be 4.85 based on the calculation made in MATLAB (huffmandict function). We can then get the average number of code length which would be 7. Finally, we need to find the total number of bits needed to represent the symbols. In this case, it would be 1. One ASCII symbol will have 8 bits.

Adding up the values above and multiplying them would give us (1+7+4.85) bits/symbol * 95 symbol which approximates to 1221 bits.

Suppose your code word dictionary requires X bits to send. Obviously, the Huffman coding must save us at least X bits when compressing the actual data, or else (when we add the bits necessary to represent the compressed data AND the code word dictionary) we would get no advantage overall (see figure 3). At the end of Section 4 above, you calculated the percentage by which your text data was compressed using the Huffman code <u>without</u> taking account of the code word dictionary. How big would the original text file have to be before it was worth Huffman coding, if we wanted to send the code word dictionary along with the compressed data?

The total number of bits obtained from above is 1221.
Since original text file is larger than 1221 and 1221 makes up 31.58% of the original text file. We would have to divide the total number of bits obtained by 0.3158 to get the size of the original text file. The original text file would have to be 1221 / 0.3158 which approximates to 3867 bits.

6. Sensitivity to errors

Although we said earlier that this lab has nothing to do with transmission errors, before we finish let's just spend a moment considering the consequences if there <u>were</u> bit errors in transmission.

Consider first the case where we don't use a VLC like Huffman coding. Instead we represent text as fixed length ASCII codes or, in the case of our small set of data in Section 3 above, we use fixed-length 4-bit code words. What is the consequence of a single bit error in the binary sequence that is generated?

The consequence of a single bit error in the binary sequence that is generated would be that the error could be identified as a symbol which could cause an inefficient transmission of information bits.

Now consider the case of a single bit error in a sequence of Huffman-generated VLCs. What do you think is the consequence now? Is the communication more, or less, sensitive to errors? Why? (Hint: there are several possible approaches to answer this question: you could just consider the problem theoretically and describe what you think will happen, you could take a sequence of Huffman codes making up a message from the set you generated in Section 3 above and by hand see what happens when one bit is flipped, or you could try to introduce a bit error in the encoded sequence in Matlab to see what is decoded.)

Communication would be more sensitive to errors. There could be a ripple effect if one of the bits is wrong since the bit is interrelated with other code word.

7. Conclusion

Having completed this laboratory, you have gained an understanding of code word length and symbol probabilities, the applicability of variable length code words, and that, depending on the symbol probabilities, these VLCs could result in data compression or expansion. Proper design of a VLC scheme uses the known or assumed probabilities to minimise the number of bits required to send a series of symbols.

You now also have experience with one of the best-known VLC methods: Huffman Coding. And you can appreciate the trade-offs in using Huffman coding, in terms of assumed or measured probabilities, and the transmission overhead that might be necessary if we need to communicate the code word dictionary.

Before finishing, could each student please click on the "Feedback" icon for this laboratory on Moodle, to record the average compression you achieved using the Huffman coding on your example text. This will allow us to look at the overall average result and you'll be able to see if you have a similar result to others. Clicking on "feedback" also gives an opportunity to provide some brief feedback on this laboratory exercise. (All inputs via the "feedback" icon are anonymous.)

Finally, please submit this completed report via Moodle by the stated deadline. In so doing, please be aware of the following:

- Even if you have had a mark assigned to you during the lab session, this mark will not be registered unless you have also submitted the report.
- Your mark may also not be accepted or may be modified if your report is incomplete or is identical to that of another student. (But it's OK if you jointly wrote a report with your partner and then both kept a copy.)
- By uploading the report, you are agreeing with the following student statement on collusion and plagiarism, and must be aware of the possible consequences.

Student statement:

– END –