CS 512: The Cost of Information

16:198:512

Instructor: Wes Cowan

Consider the following text:

```
Arma virumque cano, Troiae qui primus ab oris
Italiam, fato profugus, Laviniaque venit
litora, multum ille et terris iactatus et alto
vi superum saevae memorem Iunonis ob iram;
multa quoque et bello passus, dum conderet urbem,
inferretque deos Latio, genus unde Latinum,
Albanique patres, atque altae moenia Romae.
```

What is the information content of this paragraph? Unless you speak Latin it is tempting to say not much. However, we can address the question in almost a context-free fashion by asking the following question: how much data would I have to send to you, in order to communicate this passage to you? In standard ASCII encoding, each character is represented by 8 bits, which means (ignoring new lines) we could express this paragraph in 314 symbols $*$ 8 bits per symbol $= 2512$ bits. But can we do better? Noticing that there are only 28 characters in the whole text, we could make do with a 5 bit encoding (there are a lot of symbols we wouldn't have to worry about) which would potentially reduces us to a total 1570 bits.

Scanning through the text (again, ignoring new lines), there are 28 characters occurring with the following frequencies:

	47	e	31	a	27	u	25	i	21	t	20	m	17
o	17	r	17	s	14	n	12	l	11	,	9	q	8
p	5	v	5	b	5	d	4	f	3	c	3	L	3
A	2	I	2	g	2	.	1	;	1	R	1	T	1

From this table, we see that the space character occurs vastly more often than the letter T. If we could come up with an encoding that used fewer bits for the space character than it did for the letter T, (and generally required fewer bits for more frequent symbols) we could potentially reduce the total number of bits required to communicate the text from one person to another. Consider the following bitwise encoding:

	01	e	011	a	0111	u	01111	i	011111	t	0111111	m	01111111
o	001	r	0011	s	00111	n	001111	l	0011111	,	00111111	q	001111111
p	0001	v	00011	b	000111	d	0001111	f	00011111	c	000111111	L	0001111111
A	00001	I	000011	g	0000111	.	00001111	;	000011111	R	0000111111	T	00001111111

One of the nice things about this encoding is that it is 'prefix free', that is, no encoding occurs as the prefix of another encoding. What this means is that when the text is presented in encoded form, the bits can naturally be partitioned and the symbols recovered. Encoding the original text (ignoring newlines) gives:

```
0000100110111111101110100011011111001101111011111110011111101111
0110100011111101110011110010011111101000011111110011001011111011
1011010011111110111101111101000100110111110111111101111001110101111
0001110100100110111110011101000011011111101110011111011111011110111
1111001111110100011111011101111110010100010011001000111110111110000
1110111100111001111110100011111111011100011011111001111101111101111001
```

```
1111110111101101000110110011110111110111111010011111011111011111100
1001101110011111101011111110111100111110111111011110111111101011111
0011111001111101101011011111101011111101100110011011111001110101111
1101110001111110111111011101111110111100111010110111111010111001111
1011111100101000110111110100111011110001011001101111011111110100011
1011101100011011101101011111110110111111100100110110111111110100001
1011110011110010011110111111001110100100011101011111001101110111111
1000011111010111111011110011111011111011101001111111011110010011 1
1111011110110101101111110100011101100111110011111001010001011100 11
1001110111100111001111110100011110111101111111010001111110010011 1110
0011110110011011011111101011110011000111011011111110011111101011 1111
0011110001111101100110011011011111100111111101111011010001111011 1001
0011101000111111101110111111101111100100111111010000111011001111 0111
1001110101111001110001111011010001111111011101111110111110011111 1011
1101111111001111110100001001111100011101110011110111110011111110 0111
1011010001011101111100110110011100111111010111011111100111111110 0111
1011010111001111101111110111011010111111100101100111101111101110 0100
0011111100101111111011101100001111
```

Looking at the table, it is easy to parse through and see that the first character is an A, and see where the second character starts. The above message is a total of 1562 bits, which is an improvement (though not by much) on the naive 5-bit encoding we could use. Can we do better? Enter Huffman encoding.

Huffman encoding proceeds the following way (see the text for a more graphical / algorithmic description): first, take the two symbols with the lowest frequencies, in this case T and R. We build a prefix-free encoding starting by declaring that the *last digit of the encoding of* T *will be* 0*, and the last digit of the encoding of* R *will be* 1. But how do determine the *rest* of the encoding? Imagine merging T and R into a meta-symbol TR, which occurs with the combined frequency $1 + 1 = 2$. Replacing T and R with this meta-symbol TR, we can ask *"what should the encoding of* TR *be?"* This is a smaller version of the same problem we are originally presented with:

	47	e	31	a	27	u	25	i	21	t	20	m	17
o	17	r	17	s	14	n	12	l	11	,	9	q	8
p	5	v	5	b	5	d	4	f	3	c	3	L	3
A	2	I	2	g	2	TR	2	.	1	;	1		

Looking at this table, we can take the two least frequent symbols, . and ;, and declare that the last digit of the encoding of ; should be 0 and the last digit of the encoding for . should be 1. Again, we merge these two into a meta-symbol ;., and reinsert it into the table with its combined frequency $1 + 1 = 2$.

	47	e	31	a	27	u	25	i	21	t	20	m	17
o	17	r	17	s	14	n	12	l	11	,	9	q	8
p	5	v	5	b	5	d	4	f	3	c	3	L	3
A	2	I	2	g	2	TR	2	.;	2				

At this point, we look at the two least frequent symbols, the meta-symbol .; and the meta-symbol TR and declare the last digit of their encodings to be 0 and 1 respectively. Note that at this point, we have that the last *two* digits of the encoding for T are 10, the last two digits of R are 11, the last two digits of ; are 00, and the last two digits of

. are 01. We again merge, constructing a meta-symbol $.;TR$ with frequency $2+2=4$. This process iterates until all symbols and meta symbols have been merged into the final two, which (by my iteration) yields **sgI;.TRqeromdAL,** which will be encoded with a 0, and **tbvpcfiln ua** which will be encoded with a 1. Unpacking all this yields the full encoding for each character as

	110	e	001	a	1111	u	1110	i	1010	t	1000	m	0110
o	0101	r	0100	s	0000	n	10111	l	10110	,	01111	q	00011
p	100110	v	100101	b	100100	d	011100	f	1001111	c	1001110	L	0111011
A	0111010	I	0001001	g	0001000	.	00010101	;	00010100	R	00010111	T	00010110

This leads to an encoded message of:

```
01110100100011011111101001011010010011100110000011111
00011101001110111111011101010111111000010110010001011
01011110011100001111101010110100110010010100110111100
00011011111001001100101010010100000110000100110001 1
11101101010111101100111111010011111111100001011101001
10010001011001111111000010001110000001111110011101111
11100101101010111101011110001111100011101001010011011
11010100011010110101010000101010011110111111001101110
10110100011100110110101010110101100011100011000110100
00010100010010100000110101011111001110100011111000111
00000110001100011011111011010000101110100101101011000
00111010011000101001110011011000001111001100101111100
11100110001011001010100001011011000010011110101110101
10111101000001100101100100110101001001111011000010100
11001101110101101000111111000011111001010001111100011 1
00011000110100100001101101011001011101001101111000000
00111000000111111001110011100110110100111001011011011
10000101000011000110111001001001000010110011111101010
10111100111100101000100001100000011111000111001111000 01
01010000110011101111111000101001010111111000010000011
01111110000011011101011101110000111001110111111000101
01011111100110011111100111010101101001001111101111010 0
00111110001110100110111110000100001000001111110111110 0
00001111100011101111101101000111100111001100101001101 1
11010111111000010111010101101111001000101 01
```

This may not look like much by itself, but it is worth noting that this comes out to 1317 bits, a 52% reduction in size over the original ASCII encoding. That being so, it is worth asking - what is the best we could hope to achieve? Is there a lower bound on how small we could go? If we were paying attention to context, we might compress the original text as "Virgil's Aeneid, Book 1, Card 1." This is definitely an impressive compression, but unfortunately this method doesn't generalize to compressing anything that is not in Virgil's Aeneid, or anything that is not already written somewhere else in an easily referenced form. In comparison, the encoding constructed above could be used to compress and transmit *any* message (subject to it containing only the same 28 symbols as our original text here).

Taking the view of messages as sequences of independent symbols occurring with some frequency or probability (divorced from a larger cultural understanding), Shannon's Information Theory tells us that there is a lower bound

on the amount of information a message contains, summarized in terms of the *entropy*, or effective average bits per symbol. If a corpus contains N symbols, with symbol i occurring with frequency p_i, the bit entropy is defined to be

$$H(\text{corpus}) = -\sum_{i=1}^{N} p_i \log_2 p_i. \tag{1}$$

For our 'corpus' as above, we get $H \approx 4.1664$ bits per symbol, for a total over 314 symbols of 1308.25 bits. As such, the Huffman encoding is essentially optimal, the discrepancy between the observed 1317 bits and the theoretical bound of 1308.25 bits occurring being largely due to the frequencies not working out to exact powers of two. Note additionally, this demonstrates that the naive 5-bit / symbol encoding improvement on ASCII encoding is pretty close to optimal.

Questions:

1) We could also consider compressing the text by 'chunking' or blocking it into pairs of characters, [Ar][ma][v]. . ., and then treating each pair as a symbol, and encoding over this expanded set of symbols. Do that in this case - is the compression worse, better, or effectively no different, in terms of total number of bits required for the message? What is the bitwise entropy for the message when symbols are taken as pairs? In general, what would lead to blocking by pairs giving a better compression than individual symbols, and what would lead to blocking by pairs giving worse compression?

2) Suppose that you have a corpus over an N-symbol alphabet, each symbol occurring with frequency or probability p_i. How many possible pairs of symbols are there, and for any symbol pair, what is its frequency in this corpus? *Assume symbols occur independently with the indicated frequencies.*

3) In Question (2), how does the bitwise entropy of the individual symbol frequencies compare to the bitwise entropy of the pairwise symbol frequencies? What can you conclude from this?

4) Argue that for a given set of symbols, there is no encoding that can universally compress all possible messages in those symbols. Why is this not a problem, i.e., why do we still pursue these encodings when they cannot be universally effective?