Harrison Lam

Information Content of Natural Language

In this project, we attempt to analyse the extent to which English text is different from a Bernoulli source, using a given data set with samples of English texts encoded by A=1, …, Z=26. A Bernoulli source is a source where successive messages are independent.

Definition:

Let I_m = set of m messages which may be transmitted with probabilities $p_i, i = 1, \dots, m$

Source entropy $h = -\sum_{i=1}^{m} p_i \log_2 p_i$

By counting the number of times each character appears in the sample, the probabilities $p_0, \dots, p_{26}$ are estimated using $\hat{p}_i = \frac{number\ of\ times\ character\ i\ appears}{total\ number\ of\ characters}$. Using these probabilities, the entropy is estimated using the code shown below in figure 1, which gives an entropy $H(X)$ of 4.0688.

```
a = readmatrix('II-19-2-dataA.txt');
a = a(1:400,:);

prob = zeros(1,27);
logprob = zeros(1,27);
for i = 1:27
    prob(i) = sum(a(:)==i-1)/numel(a);
    if prob(i) > 0
        logprob(i) = log(prob(i))/log(2);
    else
        %If letter does not appear in the sample, set logp=0 so plogp=0
        logprob(i) = 0;
    end
end
entropy = -dot(prob, logprob);
```

Figure 1: estimating entropy

The Huffman code is then computed using the following code: (figure 2)

```
%Huffman
huffmancws = strings(1, 27);
tempindex = 1:27;
probtemp = prob;
for i = 1:26
    %Find the 2 groups with the lowest probabilities
    [minprob1, index1] = min(probtemp);
    probtemp(index1) = probtemp(index1)+2;
    [minprob2, index2] = min(probtemp);
    append1 = tempindex(1,index1,:);
    append2 = tempindex(1,index2,:);
    %Add 0s to all the elements in the lowest group
    for j = 1:length(append2)
        if append2(j) ~= 0
            huffmancws(append2(j)) = append('0', huffmancws(append2(j)));
        end
    end
    %Add 1s to all the elements in the 2nd lowest group
    for j = 1:length(append1)
        if append1(j) ~= 0
            huffmancws(append1(j)) = append('1', huffmancws(append1(j)));
        end
    end
    %Combine the groups
    probtemp(index2) = probtemp(index2) + probtemp(index1)-2;
    probtemp(index1) = 2;
    for k = 1:length(append1)
        if append1(k) ~= 0
            if isempty(find(append2 == 0, 1))
                tempindex(1,index2,length(append2)+k) = append1(k);
            else
                tempindex(1,index2,find(append2 == 0, 1)-1+k) = append1(k);
            end
        end
    end
    tempindex(1,index1,:) = 0;
end
el1 = dot(strlength(huffmancws),prob);
```

Figure 2: finding Huffman code

This gives the codewords in figure 3 and expected word length of 4.1142.

```
cws =

  1×27 string array

  Columns 1 through 12

    "11"    "0100"    "011110"    "001101"    "00101"    "0000"    "011100"    "001100"    "1000"    "1001"    "0011101100"    "0011100"

  Columns 13 through 24

    "00011"    "001111"    "1010"    "0110"    "011111"    "00111011011"    "00010"    "1011"    "0101"    "001000"    "00111010"    "001001"

  Columns 25 through 27

    "001110111"    "011101"    "00111011010"
```

Figure 3: Huffman codewords

This is consistent with Shannon's noiseless coding theorem, $H(X) \leq \mathbb{E}S < H(X) + 1$ where $\mathbb{E}S$ is the expected codeword length.

The Shannon-Fano code is then computed (figure 4) and compared to the Huffman code.

```
%Shannon-Fano
sfcws = strings(1, 27);
sflength = ceil(-logprob);
%Order the lengths from lowest to highest
[sflength, orderstat] = sort(sflength);
for i = 1:27
    %Loop through all the words of desired length
    for k = 1:2^sflength(i)
        flag = 0;
        bin = dec2bin(k-1,sflength(i));
        %Check if the current word is such that the code is prefix-free
        for j = 1:i-1
            str = sfcws(orderstat(j));
            if bin(1:strlength(str)) == str
                flag = 1;
                break
            end
        end
        if flag == 0
            sfcws(orderstat(i)) = bin;
            break
        end
    end
end
el2 = dot(strlength(sfcws),prob);
```

Figure 4: finding Shannon-Fano code

```
sfcws =

  1×27 string array

  Columns 1 through 10

    "000"    "0010"    "1010100"    "100100"    "01010"    "0011"    "100101"    "100110"    "01011"    "01100"

  Columns 11 through 19

    "10110001010"    "1010101"    "01101"    "100111"    "01110"    "01111"    "1010110"    "1011000101110"    "10000"

  Columns 20 through 27

    "10001"    "0100"    "101000"    "10110000"    "101001"    "1011000100"    "1010111"    "101100010110"
```

Figure 5: Shannon-Fano codewords

These codewords give an expected word length of 4.6136. While the expected length still satisfies the upper bound of the noiseless coding theorem $\mathbb{E}S < H(X) + 1$, it is much higher than the optimal length (4.6136-4.1142=0.4994). In particular, to encode data file A, which has 10000 entries, using the Shannon-Fano code will produce around 5000 more bits of data.

Assuming the source is Bernoulli and using segmentation, the entropy of a block of n letters is $H(X_1, \dots, X_n) = nH(X)$. Then, applying noiseless coding theorem to the random blocks:

$H(X_1, \dots, X_n) \leq \mathbb{E}S_n < H(X_1, \dots, X_n) + 1$ where $\mathbb{E}S_n$ is the expected codeword length representing a block of n letters

$\Rightarrow nH(X) \leq n\mathbb{E}S < nH(X) + 1$

$\Rightarrow H(X) \leq \mathbb{E}S < H(X) + \frac{1}{n}$

Hence, segmentation lowers the upper bound on the expected word length for both Huffman and Shannon-Fano code, from $H(X) + 1$ to $H(X) + \frac{1}{n}$. As the expected length of the Huffman code (4.1142) is quite close to the lower bound (4.0688), it might require segmentation of very large blocks for the expected length to have a marginal improvement. For the Shannon-Fano code, grouping messages in blocks of 2 is enough to reduce the expected length (4.6136), since the new upper bound $H(X) + \frac{1}{2} = 4.5688 < 4.6136$. This produces at least 450 fewer bits of data when encoding this data file.

Is English text Bernoulli?

The letters that appear in a single word are not independent, and some letters are appear more frequently than others right after a particular letter. In particular, using the following code, shown in figure 6, the probability that each letter appears right after a particular letter is estimated and compared to the general probability.

```
%Counting #times each letter appears right after letterIndex appears
letterIndex = 9;
afterletter = [];
for i = 1:9999
    if b(i) == letterIndex
        afterletter = [afterletter b(i+1)];
    end
end
prob2 = zeros(1,27);
for i = 1:27
    prob2(i) = sum(afterletter(:)==i-1)/numel(afterletter);
end
plot(0:26,prob);
hold on
plot(0:26,prob2)
ylabel('Probability')
xlabel('Letter');
legend('General', append('After letter ', num2str(letterIndex)));
```

Figure 6: estimating the probability each letter appears right after a particular letter

This gives the result shown below in figure 7.1, which shows that right after the letter ‘T’, the probability that an ‘H’ appears increases from 0.05 to over 0.35.

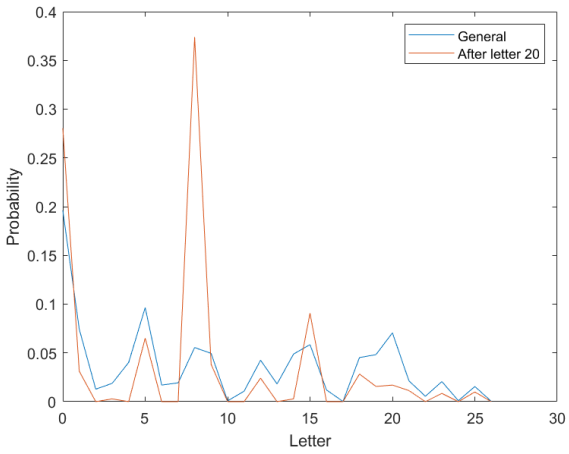

Figure 7.1: after letter 'T'

This also occurs for other letters. For example, there is a high chance the character right after the letter ‘D’ is a space (p_0 = 0.2 -> 0.65, figure 7.2), the character after an ‘M’ or ‘R’ is a lot more likely to be an ‘E’ or ‘O’ (figure 7.3, 7.4).

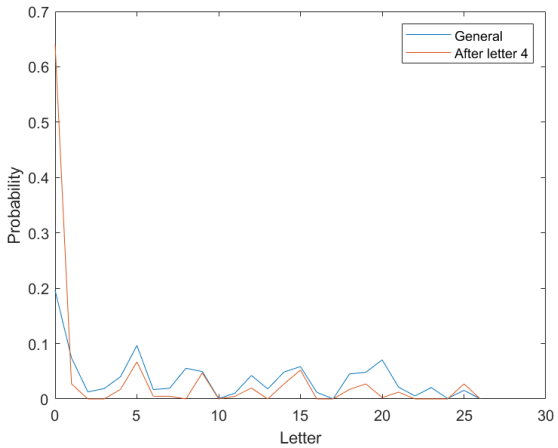

Figure 7.2: after letter 'D'

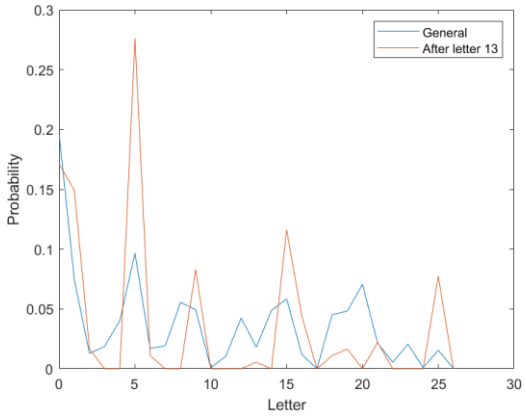

Figure 7.3: after letter 'M'

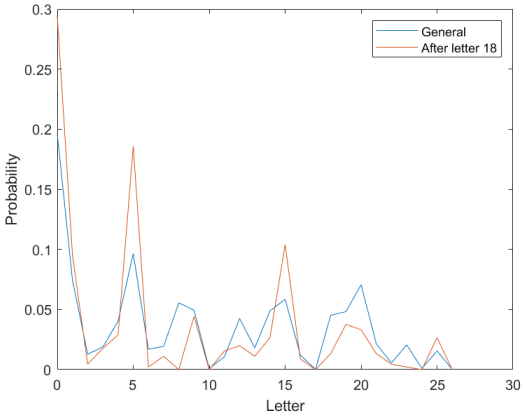

Figure 7.4: after letter 'R'

On the other hand, although harder to quantify, the words that appear in a sentence are also not independent, as the context matters and some words are lot more likely to appear in certain contexts. At the same time, it is very rare or impossible for some characters to appear multiple times in a row (e.g. space) in English text, which is not reflected in the k-fold Bernoulli probabilities. This suggests that English text is not similar to Bernoulli when analysed by letter or in blocks of fixed size. However, English text might be closer to Bernoulli when analysed by word (blocks of characters between 2 spaces), as combinations of words seem to be less restrictive than combinations of letters.

By splitting the data file into 5000 pairs of characters, the probability of each pair is estimated using the code below (figure 8). Then, the same code in figure 2 is used to construct the Huffman code, which gives an expected codeword length of 7.3146.

```
%HuffmanPair
pairprob = zeros(1,729);
a = reshape(a',2,5000);
for i = 1:27
    for j = 1:27
        paircount = (a==[i-1;j-1]);
        %Using AND operator to check if both characters match
        paircount = sum(paircount(1,:)&paircount(2,:));
        pairprob(27*(i-1)+j) = paircount/5000;
    end
end
```

Figure 8: estimating pair probabilities

However, a lot of the pairs do not appear in the data file and have an estimated probability of 0. As a result, the codewords produced for those pairs are extremely long (over 300 bits long) and it likely would not perform well when tested against a larger sample. Therefore, we will use the estimate $\hat{p}_{[i,j]} = \hat{p}_i \hat{p}_j$ instead of 0, where $\hat{p}_i$ is the general estimated probability for the i-th character, and then rescale all the probabilities so that they sum up to 1 (figure 9). This gives the pair codewords shown in figure 10.

```
%HuffmanPair
pairprob = zeros(1,729);
a = reshape(a',2,5000);
for i = 1:27
    for j = 1:27
        paircount = (a==[i-1;j-1]);
        %Using AND operator to check if both characters match
        paircount = sum(paircount(1,:)&paircount(2,:));
        if paircount > 0
            pairprob(27*(i-1)+j) = paircount/5000;
        else
            %Estimate 0 probability with Bernoulli probability
            pairprob(27*(i-1)+j) = prob(i)*prob(j);
        end
    end
end
%Rescaling probabilities
pairprob = pairprob/sum(pairprob);
```

Figure 9: replacing zero probabilities with Bernoulli probabilities

Variables - huffmanpaircws

huffmanpaircws

27x27 string

	1	2	3	4	5	6	7
1	1	0111011	0001010110101	11010011	000111	10000	00000100
2	10	01000010	00010010111	101110001	00010010110	10100010	0110110000
3	000	111011110	0001010110100	110001110000	10100101100	0101001111	0000000001110
4	111	00100111	110001110001	001010101111	00100110010	1011101001	010011111010
5	000	010011110	10100101101	00100110011	000100101001	0110000	00111111111
6	11110	1111010	0001010001	100011111	010011100	101110000	0110101111
7	001	000101001010	0000000001111	010011111011	01000011100	1011101000	1011011101
8	1101	0111001000	101110110001	001010101110	000100101000	00010010011	010001111111
9	11	10101000	00110000101	010011101	011111011	0001000100111	1100011110
10	11	01000101	000101001000	000100101010	0001000011	01101100100	0110101110
11	0100111	11000111001110	01010010010101...	11000111001111...	101110101100101	01010011101100	11101111110111...
12	0010101	01110010100	0111110011111	000100110	000100100001	0001000100110	0010101011001
13	101	00111100	100100100	111011100	01101100110	11101100	000100100000
14	101	0001010100	110100101000	010000111011	00101010011	1011011111	010101000100
15	10011	110010	01010011100	1101001011	0110111000	1101011	00000110000
16	010	10010011	111011101	001110011	011011010	00110010	111010111
17	0000	000101001001	0101010001010	111011111100	10111011011	01101011011	0000011100100

Figure 10: Huffman pair codewords

This gives an expected codeword length of 7.8324. Although this is higher than before (7.3146), it will likely result in a lower total length when using it to encode larger data files. Using segmentation, these codewords to encode the data file produces 37903 bits of data, 3200 fewer bits compared to the single-letter Huffman code. Although this is a significant improvement from the original Huffman code already, it could be improved further if the there is more data. There were only 5000 samples, which is less than 7 per pair and is not enough to obtain a good estimate of the pair probabilities.

To compare the effect of segmentation between English text and a Bernoulli source, assume that the general estimated probabilities in figure 1 are the true values and generate random data of length 10000 from a Bernoulli source using the code shown below in figure 11:

```
%Random generated text
cprob = cumsum([0, prob]);
[~, ~, randomtext] = histcounts(rand(1,10000),cprob);
randomtext = randomtext-1;
randomtext = reshape(randomtext,2,5000);
```

Figure 11: generating random text

We already have the single letter Huffman code for these probabilities from figure 3, so computing the pair Huffman code with the Bernoulli pair probabilities using the same method as before and encoding the data file gives the following bit lengths:

```
>> main

singlelength =

       41015


pairlength =

       40706
```

This shows that using segmentation on a Bernoulli source is not as effective as English text. This is because the expected length of the original Huffman code is close to the lower bound already and for a Bernoulli source, segmentation only lowers the upper bound:

$$H(X) \leq \mathbb{E}S < H(X) + 1 \quad \rightarrow \quad H(X) \leq \mathbb{E}S < H(X) + \frac{1}{2}$$

For English text, consecutive characters are not independent, so we have

$$H(X_1, X_2) < H(X_1) + H(X_2) = 2H(X)$$
$$\Rightarrow \frac{H(X_1, X_2)}{2} < H(X)$$

So, segmentation lowers both the upper bound and lower bound, and as a result, it has a larger effect on English text compared to a Bernoulli source:

$$H(X) \leq \mathbb{E}S < H(X) + 1 \quad \rightarrow \quad \frac{H(X_1, X_2)}{2} \leq \mathbb{E}S < H(X) + \frac{1}{2}$$

However, using segmentation on English text will require a bigger sample to obtain a more accurate estimate of the probabilities. For example, using the given data file with 10000 entries, there are only 5000 data points for 27^2=729 outcomes, which is not enough to obtain a good estimate for the probabilities. For a Bernoulli source, we can estimate the single letter probabilities quite accurately as there are 10000 entries for 27 outcomes, then use the product of single letter probabilities as the pair probabilities.

Reference:

https://www.maths.cam.ac.uk/undergrad/catam/II/19pt2.pdf