Hand-in 2

Zhongwang Fu, Personnumber: 200203237532, 2025.2.10

1. Code table

When each new line is represented by two characters: a carriage return ('\r') and a line feed ('\n'), each occurring 3608 times. A total of 74 unique characters are present in the dataset. The given text file consists of 152089 characters, and the counts and probabilities of the characters are shown in table 1.

We can see that high-frequency characters, such as space (' '), were assigned the shortest codewords. For instance, space uses a 2 bits code (11). In contrast, low-frequency characters like '\x1a' and '2' received much longer codewords (17 bits), corresponding to the principle of minimizing average code length by prioritizing frequent symbols.

The entropy of the estimated distribution, calculated as $H(X) = -\sum p(x)\log_2 p(x) = 4.567680\ \text{bits/character}$, representing the theoretical lower bound for lossless compression. The constructed Huffman code achieved an average codeword length $L = \sum p(x) l_x = 4.612444\ \text{bits/character}$, which is very close to the entropy, confirming that our Huffman code is close to the theoretical limit but still respects integer constraints. Thus, we can conclude that $H(X) \le L < H(X) + 1$, which aligns with the optimal efficiency of Huffman coding.

Tabel 1. Huffman code of each character using 2 characters for new line

Character	Count	Probability	Huffman Code	$\lceil -\log(p(x)) \rceil$	l_x
'\x1a'	1	0.000007	10100011001101111	18	17
'2'	1	0.000007	10100011001101110	18	17
'9'	1	0.000007	10100011001101101	18	17
'Z'	1	0.000007	10100011001101100	18	17
'['	2	0.000013	101000110011111	17	15
']'	2	0.000013	101000110011110	17	15
'X'	4	0.000026	101000110011010	16	15
'_'	4	0.000026	10100011001110	16	14
'J'	8	0.000053	10100011001100	15	14
'V'	42	0.000276	101000110010	12	12
')'	55	0.000362	000100110001	12	12
'('	56	0.000368	000100110000	12	12
'*'	60	0.000395	10100011101	12	11
'P'	64	0.000421	10100011100	12	11

'U'	66	0.000434	10100011000	12	11
'F'	74	0.000487	01100101011	12	11
'z'	77	0.000506	01100101010	11	11
'G'	82	0.000539	01100101001	11	11
'K'	82	0.000539	01100101000	11	11
'Q'	84	0.000552	00111100111	11	11
'B'	91	0.000598	00111100110	11	11
'L'	98	0.000644	00010011001	11	11
'"'	113	0.000743	00010010101	11	11
'Y'	114	0.000750	00010010100	11	11
'N'	120	0.000789	1010001111	11	10
'q'	125	0.000822	1010001101	11	10
'j'	138	0.000907	1010001011	11	10
'R'	140	0.000921	1010001010	11	10
'C'	144	0.000947	0110010111	11	10
'x'	144	0.000947	0110010110	11	10
'O'	176	0.001157	0011110010	10	10
'E'	188	0.001236	0011110001	10	10
'D'	192	0.001262	0011110000	10	10
';'	194	0.001276	0001001111	10	10
'M'	200	0.001315	0001001110	10	10
'?'	202	0.001328	0001001101	10	10
'S'	218	0.001433	0001001011	10	10
':'	233	0.001532	101010111	10	9
'W'	237	0.001558	101010110	10	9
'H'	284	0.001867	101000100	10	9
'!'	449	0.002952	000100100	9	9
'T'	472	0.003103	10101010	9	8
'A'	638	0.004195	01100100	8	8
'-'	669	0.004399	00111101	8	8
'I'	733	0.004820	00111001	8	8
'v'	803	0.005280	00111000	8	8
'.'	977	0.006424	1010100	8	7
'k'	1076	0.007075	1010000	8	7
'`'	1108	0.007285	0110011	8	7

'b'	1383	0.009093	0011111	7	7
'p'	1458	0.009586	0011101	7	7
'"'	1761	0.011579	0001000	7	7
'm'	1907	0.012539	101011	7	6
'f'	1926	0.012664	101001	7	6
'y'	2150	0.014136	011111	7	6
'c'	2253	0.014814	011110	7	6
','	2418	0.015899	011000	6	6
'w'	2437	0.016024	010111	6	6
'g'	2446	0.016083	010110	6	6
'u'	3402	0.022368	000101	6	6
'\n'	3608	0.023723	10111	6	5
'\r'	3608	0.023723	10110	6	5
'l'	4615	0.030344	01110	6	5
'd'	4739	0.031159	01101	6	5
'r'	5293	0.034802	01010	5	5
's'	6277	0.041272	00110	5	5
'i'	6778	0.044566	00011	5	5
'n'	6893	0.045322	00001	5	5
'h'	7088	0.046604	00000	5	5
'o'	7965	0.052371	1001	5	4
'a'	8149	0.053580	1000	5	4
't'	10212	0.067145	0100	4	4
'e'	13381	0.087981	0010	4	4
' '	28900	0.190020	11	3	2

In my operating system, Windows 11, and in Python 3.9, a new line is represented by a single character ('\n') by default. Then the total number of characters in the dataset is 148481, compared to 152089 in the previous case. This difference affects the character frequencies and probability distribution, leading to some variations in the Huffman codes as shown in table 2.

The entropy of the estimated distribution is $H(X) = 4.512877\ bits/character$, representing the theoretical lower bound for lossless compression. The constructed Huffman code achieves an average codeword length of $L = 4.555290\ bits/character$.

Tabel 2. Huffman code of each character using 1 characters for new line

Character	Count	Probability	Huffman Code	$\lceil -log(p(x)) \rceil$	l_x
'\x1a'	1	0.000007	01111011001101111	18	17
'2'	1	0.000007	01111011001101110	18	17
'9'	1	0.000007	01111011001101101	18	17
'Z'	1	0.000007	01111011001101100	18	17
'['	2	0.000013	011110110011111	17	15
']'	2	0.000013	011110110011110	17	15
'X'	4	0.000027	011110110011010	16	15
'_'	4	0.000027	01111011001110	16	14
'J'	8	0.000054	01111011001100	15	14
'V'	42	0.000283	011110110010	12	12
')'	55	0.000370	000010110001	12	12
'('	56	0.000377	000010110000	12	12
'*'	60	0.000404	01111011101	12	11
'P'	64	0.000431	01111011100	12	11
'U'	66	0.000445	01111011000	12	11
'F'	74	0.000498	01011101011	11	11
'z'	77	0.000519	01011101010	11	11
'G'	82	0.000552	01011101001	11	11
'K'	82	0.000552	01011101000	11	11
'Q'	84	0.000566	00110100111	11	11
'B'	91	0.000613	00110100110	11	11
'L'	98	0.000660	00001011001	11	11
'"'	113	0.000761	00001010101	11	11
'Y'	114	0.000768	00001010100	11	11
'N'	120	0.000808	0111101111	11	10
'q'	125	0.000842	0111101101	11	10
'j'	138	0.000929	0111101011	11	10
'R'	140	0.000943	0111101010	11	10
'C'	144	0.000970	0101110111	11	10
'x'	144	0.000970	0101110110	11	10
'O'	176	0.001185	0011010010	10	10

'E'	188	0.001266	0011010001	10	10
'D'	192	0.001293	0011010000	10	10
';'	194	0.001307	0000101111	10	10
'M'	200	0.001347	0000101110	10	10
'?'	202	0.001360	0000101101	10	10
'S'	218	0.001468	0000101011	10	10
':'	233	0.001569	101000111	10	9
'W'	237	0.001596	101000110	10	9
'H'	284	0.001913	011110100	10	9
'!'	449	0.003024	000010100	9	9
'T'	472	0.003179	10100010	9	8
'A'	638	0.004297	01011100	8	8
'-'	669	0.004506	00110101	8	8
'I'	733	0.004937	00110001	8	8
'v'	803	0.005408	00110000	8	8
'.'	977	0.006580	1010000	8	7
'k'	1076	0.007247	0111100	8	7
'`'	1108	0.007462	0101111	8	7
'b'	1383	0.009314	0011011	7	7
'p'	1458	0.009819	0011001	7	7
'"'	1761	0.011860	0000100	7	7
'm'	1907	0.012843	101001	7	6
'f'	1926	0.012971	011111	7	6
'y'	2150	0.014480	011101	7	6
'c'	2253	0.015174	011100	7	6
','	2418	0.016285	010110	6	6
'w'	2437	0.016413	010101	6	6
'g'	2446	0.016473	010100	6	6
'u'	3402	0.022912	000011	6	6
'\n'	3608	0.024299	10101	6	5
'l'	4615	0.031081	01101	6	5
'd'	4739	0.031917	01100	5	5
'r'	5293	0.035648	00111	5	5
's'	6277	0.042275	00101	5	5
'i'	6778	0.045649	00100	5	5

'n'	6893	0.046423	00000	5	5
'h'	7088	0.047737	1011	5	4
'o'	7965	0.053643	1001	5	4
'a'	8149	0.054882	1000	5	4
't'	10212	0.068776	0100	4	4
'e'	13381	0.090119	0001	4	4
' '	28900	0.194638	11	3	2

2. Solution explanation

The program first processes a given text file to analyze character distributions. The file is read using the readTxt(path, newline_representation=2) function, which reads the text file and calculates character counts and probabilities. In my system, Python on Windows (Win11, Python 3.9) represents new lines with a single '\n', leading to a total character count of 148481 in this case. This differs from systems where new lines are stored as "\r\n", resulting in some variations in frequency distributions and Huffman encodings. To deal with that, if newline_representation == 1, the function removes '\r' (carriage return) since Python typically represents a newline as a single '\n' on modern systems. If newline_representation == 2, it ensures '\r' has the same count as '\n', treating "\r\n" as two separate characters.

The Huffman codes are generated using the buildHuffmanCodes(probabilities) function, which employs a priority queue to iteratively merge the least probable characters. Initially, each character and its probability are stored in a heap, ensuring that the elements with the smallest probabilities are processed first. In each step, the two nodes with the smallest probabilities are popped from the heap. Each character in the lower-probability vector is coded '1' as a prefix, while characters in the higher-probability vector are coded '0'. At last, merge the two character vectors and add up their probabilities, and then push merged node to the queue. The process continues until only final node remains, and return the constructed Huffman codes.

At last, I calculate the entropy of the character distribution $H(X) = -\sum p(x)\log_2 p(x)$ and the average code length $L = \sum p(x) l_x$. The result should satisfy $H(X) \leq L < H(X) + 1$.

Appendix(Python 3.9)

```
import heapq
import math


def readTxt(path, newline_representation=1):
    with open(path, 'r') as file:
        text = file.read()

    char_count = {}

    for char in text:
        char_count[char] = char_count.get(char, 0) + 1

    if newline_representation == 1:
        char_count.pop('\r', None)
    elif newline_representation == 2:
        char_count['\r'] = char_count.get('\n', 0)

    total_chars = sum(char_count.values())

    char_probability = {char: count / total_chars for char, count in char_count.items()}

    return char_count, char_probability


def buildHuffmanCodes(probabilities):
    heap = [[prob, [char]] for char, prob in probabilities.items()]
    heapq.heapify(heap)

    codes = {}

    while len(heap) > 1:
        low = heapq.heappop(heap)
        high = heapq.heappop(heap)
```

```
        for char in low[1]:
            codes[char] = '1' + codes.get(char, '')
        for char in high[1]:
            codes[char] = '0' + codes.get(char, '')

        merged_prob = low[0] + high[0]
        merged_chars = low[1] + high[1]
        heapq.heappush(heap, [merged_prob, merged_chars])

    return codes


if __name__ == '__main__':
    path = './Alice29.txt'
    count, probabilities = readTxt(path=path, newline_representation=2)

    huffmanCodes = buildHuffmanCodes(probabilities)

    print(
        f"{'Character':<10} {'Count':<5} {'Probability':<12} {'Huffman Code':<18}
{'l=Upper(-log(p))':<10} {'Code Length':<10}")
    for char, code in huffmanCodes.items():
        prob = probabilities[char]
        upper_log_p = math.ceil(-math.log2(prob))
        code_length = len(code)
        cnt = count[char]
        print(f"{repr(char):<10}      {cnt:<5}      {prob:<12.6f}      {code:<18}
{upper_log_p:<10} {code_length:<10}")

    entropy = 0
    for prob in probabilities.values():
        entropy -= prob * math.log2(prob)

    avgLength = 0
```

```
for char, prob in probabilities.items():
    avgLength += prob * len(huffmanCodes[char])

print(f"Entropy (H(X)): {entropy:.6f} bits")
print(f"Average Code Length (L): {avgLength:.6f} bits")

assert ((entropy <= avgLength) & (avgLength < entropy + 1))
```