

情報理論 最終レポート1

Page • 1 反方向連結 • Tag • 二月 10, 2025

問題一

今回のレポートは英語の文章を一つ選んで、そしてそれをハフマン符号化をしたら、その結果を観察することである。

作ったプログラムと使った英語文章は全て [Github](#) にアップロードした

| https://github.com/Koios1143/Information_Theory_HW1

実装の詳細解説

Step1: エントロピーを計算

まずはエントロピーを計算することである。なので、文章の中にどんな文字があるか、いくらあるか、出現頻度はどれぐらいかなどを先に求めないといけないのだ。

文章ファイルを読み込んだら、文章に出てきた各文字の出現回数を `char_counts` に記録して、さらに出現回数によりソートすると、 `char_counts_lst` になった。

```
Python ▾  
"""  
Calculate character counts and total characters  
"""  
total_chars = len(text)  
char_counts = {}  
for char in text:  
    if char not in char_counts:  
        char_counts[char] = 1  
    else:  
        char_counts[char] += 1  
  
char_counts_lst = sorted(char_counts.items(), key=lambda x: -x[1])  
print(f'{"=" * 10} Character counts {"=" * 10}')print(f'{char_counts_lst}\n')  
print(f'Total characters: {total_chars}')
```

各文字の出現回数があると、出現頻度を求めることができるのだ。下の通りに `char_freq` と頻度によりソートした `char_freq_lst` がもらえるのだ。

```
Python ▾  
"""  
Calculate character frequency based on character counts and total characters  
"""  
char_freq = {}  
for char, counts in char_counts_lst:  
    char_freq[char] = counts / total_chars  
char_freq_lst = sorted(char_freq.items(), key=lambda x: -x[1])  
output_freq_lst = [(key, f'{value:.5f}')] for key, value in char_freq_lst]  
print(f'{"=" * 10} Character frequency {"=" * 10}')print(f'{output_freq_lst}\n')
```

各文字の出現頻度があると、エントロピーを求めることができるのだ。

Python ▾

```
"""
Calculate entropy
"""
entropy = 0
for char, prob in char_freq_lst:
    entropy -= prob * log2(prob)
print(f'Entropy: {entropy}')
```

Step 2：ハフマン符号化

各文字の出現頻度があると、符号の木が作れる、つまりハフマン符号化することができるのだ。

符号の木を作る流れは、毎回各記号の中に、出現頻度が一番低い記号を二つまとめて、新たな記号とみなし、その記号の出現頻度がもとの二つの記号の出現頻度の和とする。この操作は全ての記号が一つの記号とみなされるまで繰り返すのだ。

毎回出現頻度が一番低い記号を二つ取るため、Heapを使って実装するはよいと思うのだ。

Python ▾

```
"""
Huffman Code
"""

child = [] # (left_child, right_child), leaf node if left_child == right_child
heap = [] # (prob, idx)
tbl = [] # char lookup table
for node, prob in char_freq.items():
    idx = len(child)
    child.append((idx, idx))
    heappush(heap, (prob, idx))
    tbl.append(node)

while heap is not None and len(heap) >= 2:
    left = heappop(heap)
    right = heappop(heap)
    heappush(heap, (left[0] + right[0], len(child)))
    child.append((left[1], right[1]))
```

符号の木が作ったら、木の根から深さ優先探索(Depth-First Search)を実行すると、各文字のハフマン符号を分かるだろう。

Python ▾

```
### Build huffman code from tree
huff_code = {}
def build_huff_code(idx: int, code: str):
    if(child[idx] == (idx, idx)):
        # leaf node
        huff_code[tbl[idx]] = code
    else:
        build_huff_code(child[idx][0], code+'0')
        build_huff_code(child[idx][1], code+'1')
build_huff_code(len(child)-1, '')
output_huff_code = {key: value for key, value in sorted(huff_code.items(), key=lambda x:len(x[1]))}
print(f'{"=" * 10} Huffman code {"=" * 10}')
print(output_huff_code)
```

Step 3：平均符号語長を求める

符号化したら、符号語長と符号の出現頻度により、平均符号語長を求めることができるのだ。

Python ▾

```
"""
Mean code length
"""

huff_code_lst = sorted(huff_code.items(), key=lambda x:x[0])
mean_code_length = 0
for char, code in huff_code_lst:
```

```
mean_code_length += len(code) * char_freq[char]
print(f'Mean code length: {mean_code_length}')
```

Step 4：ファイルのサイズの変化

元々の文章は英語、数字、そして読点等の記号で書いたもので、その一方、ハフマン符号化した各符号は0と1の組み合わせである。なので、同じ0と1の組み合わせで比べる、あるいは同じ ASCII Charactersの組み合わせで比べる、という二つの方法があるのだ。

では、まず、本来の文章の各文字を8ビットに変換しよう。

```
Python ▾
"""
How it looks like if we're coding the original transcript with bunch of 8-bits 0/1?
"""

encode = {}
decode = {}
for char in char_counts:
    encode[char] = '{:08b}'.format(ord(char))
    decode['{:08b}'.format(ord(char))] = char
coded_text = ''
with open('original_binary_coded.txt', 'w+') as f:
    for char in text:
        f.write(encode[char])
        coded_text += encode[char]
```

そして、ハフマン符号化した文章もファイルに書き込み。

```
Python ▾
"""
How it looks like if we apply huffman code?
"""

huff_bin_text = ''
with open('huffman_binary_coded.txt', 'w+') as f:
    for char in text:
        f.write(huff_code[char])
        huff_bin_text += huff_code[char]
```

そうすると、ファイルのサイズを比べることができるのだ。

```
Python ▾

original_binary_file_size = os.stat('original_binary_coded.txt').st_size
huffman_binary_file_size = os.stat('huffman_binary_coded.txt').st_size
original_binary_size = len(coded_text)
huffman_binary_size = len(huff_bin_text)
print(f"Original binary code\tfile size {original_binary_file_size}\t\tlength: {original_binary_size}")
print(f"Huffman binary code\tfile size {huffman_binary_file_size} [green]({ratio(original_binary_file_size,
huffman_binary_file_size):.2f}% ↓)[/green]\tlength: {huffman_binary_size} [green]({(original_binary_size-
huffman_binary_size)/original_binary_size*100:.2f}% ↓)[/green]")
```

そして、ハフマン符号化した文章を読み込んで、8ビットつつASCII Codeに変換しよう。

```
Python ▾
"""
How it looks like if we code the binary code into characters?
"""

huff_text = ''
with open('huffman_text.txt', 'w', encoding="iso-8859-1") as f:
    with open('huffman_binary_coded.txt', 'r') as g:
        while True:
            chunk = g.read(8)
            if not chunk:
                break
```

```
f.write(chr(int(chunk, 2)))
huff_text += chr(int(chunk, 2))
```

そうすると、本来の文章のファイルと比べることができるのだ。

Python ▾

```
original_file_size = os.stat('transcript.txt').st_size
huffman_file_size = os.stat('huffman_text.txt').st_size
original_text_size = len(text)
huffman_text_size = len(huff_text)
print(f"Original text\t\t\tfile size {original_file_size}\t\t\tlength: {original_text_size}")
print(f"Huffman text\t\t\tfile size {huffman_file_size} [green]({ratio(original_file_size,
huffman_file_size):.2f}% ↓)[/green]\tlength: {huffman_text_size} [green]({ratio(original_text_size,
huffman_text_size):.2f}% ↓)[/green]")
```

補足： `ratio()` の定義は以下の通りである。

Python ▾

```
def ratio(orig, new):
    return (orig - new) / orig * 100
```

実験結果

- この文章は 17766 文字である
- エントロピーは 4.452240038286298 である
- ハフマン符号化の結果は下に表している

Python ▾

```
{
    'e': '000',          ' ': '111',
    'r': '0011',         'i': '0101',
    'n': '0110',         'o': '0111',
    't': '1001',         'a': '1010',
    'u': '00101',        'd': '01001',
    'h': '10000',        'l': '11001',
    's': '11011',        'b': '001000',
    ',': '010000',       '.': '010001',
    'f': '100011',       'g': '101100',
    'y': '101101',       'w': '101111',
    'p': '110000',       'm': '110001',
    'c': '110101',       'A': '1000101',
    'v': '1101000',      '\n': '1101001',
    'I': '00100100',     'T': '00100110',
    '(': '10001000',     ')': '10001001',
    'k': '10111011',     '"': '001001010',
    '-': '001001110',    'W': '101110011',
    'M': '0010010111',   'C': '0010011110',
    'z': '0010011111',   'P': '1011100001',
    'S': '1011100101',   'x': '1011101011',
    'E': '00100101100',  '0': '10111000000',
    'F': '10111000001',  'D': '10111000100',
    'N': '10111000101',  'G': '10111000110',
    'B': '10111010000',  'j': '10111010001',
    'O': '10111010010',  'U': '10111010100',
    'V': '001001011011', 'R': '101110001110',
    'H': '101110001111', ':': '101110010000',
    'K': '101110010001', 'J': '101110100110',
    'q': '101110100111', '2': '101110101010',
    'L': '101110101011', 'Y': '0010010110100',
    '5': '0010010110101', '1': '1011100100100',
    '7': '1011100100101', '8': '1011100100110',
```

```
'9': '10111001001110',  '3': '10111001001111'
}
```

- 平均符号語長は 4.482888663739727 である
- 二つの比べる方法があるが、同じく 43.96% のサイズを圧縮した


Python ▾

Original text	file size 17766	length: 17766
Huffman text	file size 9956 (43.96% ↓)	length: 9956 (43.96% ↓)
Original binary code	file size 142128	length: 142128
Huffman binary code	file size 79643 (43.96% ↓)	length: 79643 (43.96% ↓)

- 情報源符号化定理を満たすことが確認した ($\epsilon = 1$ と仮定する)

$$H(X) \leq \frac{\bar{L}}{n} < H(X) + \epsilon$$
$$\Rightarrow 4.452240038286298 \leq 4.482888663739727 < 5.452240038286298 \quad \blacksquare$$

問題二



1 情報記号あたり確率0.01で誤りが発生する二元対象通信路を考えます。

この通信路に、m=2,3,4,5のハミング符号で通信路符号化を行ってデータの送信を行う場合、情報伝送速度（符号のレート）Rと復号誤り率Peがそれぞれどれくらいの値となるかを調べてください。また、通信路符号化定理を踏まえ、同じ情報伝送速度で別の符号を構成する場合どれだけの改善の余地がありうるかを評価してください。

まず、検査記号の長さ m に対して、ハミング符号の符号語長が $n = 2^m - 1$ 。したがって、情報記号の桁数が $k = 2^m - m - 1$ となる。なので、各 m に対して n, k そして符号のレート R は以下ようになる。

m	(n, k)	$R = k/n$
2	(3, 1)	$1/3 \approx 0.333$
3	(7, 4)	$4/7 \approx 0.571$
4	(15, 11)	$11/15 \approx 0.733$
5	(31, 26)	$26/31 \approx 0.839$

次は復号誤り率を求めよう。ハミング符号は1ビット誤り訂正可能なため、2ビット以上の誤りが発生すると誤り訂正に失敗し、復号誤りが生じる。 i ビット誤りが発生する確率は：

$$P(i) = \binom{n}{i} p^i (1 - p)^{n-i}$$

ここに、 $p = 0.01$ とする。したがって、復号誤り率 P_e は：

$$P_e = \sum_{i=2}^n P(i)$$

このように計算すると、各 m に対しての復号誤り率 P_e を求めることができる。

m	P_e
2	≈ 0.000298
3	≈ 0.002031
4	≈ 0.009630
5	≈ 0.038390

計算するには、以下のプログラムでできる。

Python ▾

```

from math import comb

p = 0.01
m_values = [2, 3, 4, 5]

# Calculate P_e for Hamming codes
def calculate_pe(n, p):
    pe = sum(comb(n, i) * (p ** i) * ((1 - p) ** (n - i)) for i in range(2, n + 1))
    return pe

# Compute results
results = []
for m in m_values:
    n = 2**m - 1
    k = n - m
    R = k / n
    Pe = calculate_pe(n, p)
    results.append((m, n, k, R, Pe))

for result in results:
    print(result)

```


式 (6.16) によると、通信路容量 C は：

$$C = \max_{p(a_i)} H(Y) - H(q) = 1 - H(q)$$

この例には $q = 0.01$ なので、通信路容量 $C = 1 - H(0.01) = 1 - (-0.01 \times \log_2 0.01) \approx 0.93356$ になる

通信路符号化定理によると、 $R < C$ ならば、任意に小さい復号誤り率で送信できる通信路符号が存在する。前の計算の結果により、全ての R は通信路容量 C より小さいため、全ての m に対して、任意に小さい復号誤り率で送信できる通信路符号が存在することが分かる。

問題三



授業中の説明の中で、わかりづらかった点はなにか。それをどう克服して理解したかを述べてください。また、その内容について、情報理論の初学者が理解しやすい説明を、自分なりの方法で記してください。図などを活用してもよいです。

授業中によくわからないのは Hamming Code のパリティ検査行列 H だった。授業の時に、その行列を作るには、 $1 \sim 2^m - 1$ の十進法数字を長さ m の二進法数字に変換して、順番に行列の column に入ればよい、ということをつかったが、その理由は分からなかった。

では、一体どうしてこのように行列を作るのか？

まず、今は「1ビット誤りの訂正が可能」という Hamming Code を前提として考えているのだ。

できるだけ簡単にエラーが発生するビットを見つけるように、パリティ検査行列 H と受信した記号列と行列の乗法をしたら、syndrome という結果から、エラービットを表す性質が欲しいということだ。

長さ n の情報で 1 ビット誤りが発生するには、 n 種類の誤りがあるのだ。(1 番目のビットが誤り、2 番目のビットが誤り、...、 n 番目のビットが誤り)

なので、「 n 種類の vector が必要」、「syndrome は n 種類が必要」という二つの条件があるのだ。

- n 種類の vector が必要

なので、パリティ検査行列の column の数量は n であり、各 vector は異なるである。

- syndrome は n 種類が必要

なので、パリティ検査行列の row の数量は $\sqrt{n+1}$ である

ここまでパリティ検査行列はどうして $n \times \sqrt{n+1}$ の型であり、各 column は異なることを説明したが、「どうして 1 から $2^m - 1$ まで順番に行列に入れるか?」、「どうして $Hx = O$ を満たす全ての x が Hamming Code を満たすのか?」はまだ説明していない。

「どうして 1 から $2^m - 1$ まで順番に行列に入れるか?」という問題について、実は別にこのように順番に入れなくても大丈夫けれど、順番に入れれば、syndrome を十進法数字に変換したら直ぐにエラービットであるし、実に便利で理解しやすいから、普通はそうやるのだ。

「どうして $Hx = O$ を満たす全ての x が Hamming Code を満たすのか?」という問題について、少しだけ線型代数の知識が必要だ。

今考えている Hamming Code は「1 ビット誤りの訂正が可能」という前提がある。なので、Hamming Code を満たす各記号の Minimum Hamming Distance は 3 である。

なら、少なくとも三つの vector から zero vector を作らなければならないのだ。そうではないならば、Minimum Hamming Distance は 3 以下の可能性があるになってしまうのだ。

行列の各 column に異なる二進法数字を入れたという作り方は、直ぐにこの条件を満たすのだ。

このように、「パリティ検査行列はどうしてそのように作るのか」のを詳しく説明した。そう考えたらちゃんと理解できると思うのだ。