情報理論　最終レポート1

Page • Tag • 今天

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

先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たら、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
"""
```

```python
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、新たの記号とみなし、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

```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

```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

```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\tfile size {original_file_size}\t\t\tlength: {original_text_size}")
print(f"Huffman  text\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 4.4828886637397 27 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）

$$H(X) \leq \frac{\bar{L}}{n} < H(X) + \epsilon$$

$$\Rightarrow \quad 4.452240038286298 \leq 4.482888663739727 < 5.452240038286298 \quad \blacksquare$$