SSIE500 Homework 4

Grant T. Aguinaldo

November 12, 2020

1 Introduction

As part of this homework assignment, code was developed to encode a piece of text using the Huffman Encoding algorithm. Once the text was encoded, the Average Codeword Length was determined and compared to the Information Entropy of the raw text.

The `Python 3.8` code that was generated for this project is included as Appendix A to this document. This document was created with Overleaf and uses `graphicx` to insert the figure images and `pdfpages` to embed the pdf document that contains the code used in this project. The code shown in the Appendix was generated using the Jupyter Notebook integrated development environment (IDE), rendered as a Tex file using functionality built into the Notebook interface, and then converted to a pdf. The resulting pdf was then included in this document using `pdfpages`.

1.1 Project Requirements

This project has several requirements that need to be met. A comparison of how each of the project requirements were met is included in **Table** 1. For clarity, the abbreviation "App" refers to the Appendix that is included in this document.

Table 1: Summary of Requirements

	Specification Section	Report Section
Read text document.	1	App. Cell 2
Generate Huffman Code.	2	App. Cell 5
Calculate Average Codeword Length.	3	App. Cell 7
Produce LaTeX report using `pdfpages`.	4	1

The code for this project is being maintained under version control using git.[1]

[1] https://github.com/grantaguinaldo/ssie/tree/master/ssie500/hw4

1.2 Data

The data for this project was obtained from the Project Gutenberg website. Project Gutenberg provides access to more than 60,000 free eBooks in a variety of formats. We decided to complete this assignment using the US Constitution. [2]

2 Analytical Approach

For this project, we largely relied on the `heaqp` library and used the imperative programming paradigm. Given the fact that the Huffman Coding scheme uses a binary search tree, we decided to use the `heaqp` library since it is able to abstract many of the inner workings of a Binary Search tree. Generally, our approach to solving this project utilized the following steps:

1. Determine the frequency of all of the characters in the text.
2. Populate a min heap containing all of the characters and frequencies.
3. Iteratively build nodes by taking the smallest two nodes from the heap.
4. At each step append a `0` or `1` depending on the depth of the tree, determine the overall frequency of the node, create a larger node.
5. Append the larger node back to the heap and restart the cycle at Step 2.
6. Once all of the characters have been processed, create a decode dictionary to reveal the character mapping.

2.1 Information Calculations

As noted in Section 3 of the project specification, we needed to calculate the Expected Information from the original (unencoded) text. To meet this requirement, we used the expression below to calculate the Expected Information.

$$H(X=k) = \sum_{k \in X} -1 \cdot \mathcal{P}(X=k) \cdot log_2 \mathcal{P}(X=k) \tag{1}$$

In addition, the project specification also requires us to calculate the Average Codeword Length of the Huffman-encoded text. To meet this requirement, we used the expression below to calculate the Average Codeword Length.

$$\text{length of encoded text / length of un-encoded text} \tag{2}$$

Moreover, since Huffman Encoding is a variable length encoding scheme where the more frequent characters have a shorter code length than the longer ones, we also calculated the Average Codeword Length using the expression below to triangulate the value that was computed earlier.

[2] http://www.gutenberg.org/cache/epub/5/pg5.txt

$$\bar{\lambda} = \sum_{k \in X} \lambda_k \cdot \mathcal{P}(X = k) \quad (3)$$

Where λ_k is the length of the k^{th} codeword provided that $k \in X$.[3] The Expected Information represents the absolute minimum value, while the Average Codeword Length represents the optimum value.

3 Results

3.1 Character Frequency

The distribution of the characters in the raw text is presented in the figure below, and follows what appears to be a power-law distribution where the empty space ' ', letter `e` and letter `t` are the most frequent letters in the text.

3.2 Information Theoretic Quantities

From the expressions noted in Equation 1 and Equation 2, the amount of Expected Information was calculated to be `4.685 bits` and the Average Codeword Length was calculated to be `4.719 bits`.

4 Discussion

Within this work, we were able to develop an approach for encoding a piece of text from the English language using a Huffman Code. From our efforts, we were able to make two observations.

First, the character counts from the text are consistent with the distribution of letters found in the English language where empty space ' ', letter `e` and letter `t` are the most frequent letters in the text.[4]

Second, the Average Codword Length of the Huffman-Encoded system satisfies Shannon Source Coding Theorem. In words, the Shannon Source Coding Theorem states that for a given set of codewords, some codewords will need to be shorter and some will need to be longer. From a compression standpoint, we want our compression algorithm to generate as few bits as possible but generate the right amount of bits to preserve the information in the original message. The Shannon Source Coding Theorem also establishes the fact that we cannot compress an information source to an amount less than the Entropy of the original system. From the results obtained, we see that the shorter codewords are indeed associated to the characters that are more frequent and the longer codewords are associated to the characters that are less frequent. In addition,

[3] https://www.cs.toronto.edu/ radford/csc310.F11/week3.pdf

[4] http://pi.math.cornell.edu/ mec/2003-2004/cryptography/subs/frequencies.html

we see that the Average Codeword Length was calculated to be `4.719 bits` using Huffman Encoding. From our work, we also observed that the expected information (Information Entropy) was calculated to be `4.685 bits`. These results are consistent with the fact that we cannot compress the text to amount that is less than the Entropy of the original system. Finally, from the results obtained herein, we see that when applying Huffman Encoding, the uncertainty in interpreting the original system has decreased by a factor of 0.034 (calculated as $4.719 - 4.685$) which equates to approximately `1 bit` of information (calculated as $2^{0.034}$).

Appendix A

November 12, 2020

[1]:
```
#Standard Imports
import pandas as pd
from collections import Counter
import requests as r
import matplotlib.pyplot as plt
import heapq
import numpy as np
%matplotlib inline
```

[2]:
```
#URL of text for US Constitution from `gutenberg.org`
url = 'http://www.gutenberg.org/cache/epub/5/pg5.txt'
#Get text and store in an object.
text = r.get(url).text.strip().replace('\n', '')

#Develop char counts from the text
textDict = dict(Counter(text))

#Store results in a dataframe
df = pd.DataFrame({'char': list(textDict.keys()), 'char_count': list(textDict.
 ↪values())})
df['freq'] = df['char_count'] * (1/df.char_count.sum())
df = df.sort_values(by='freq', ascending=False).reset_index(drop=True)

#Store results in a dict
freq = {row['char']:row['freq'] for index, row in df.iterrows()}
```

[3]:
```
print('The First 2,000 Characters of the raw text: \n---')
text[:2000]
```

```
The First 2,000 Characters of the raw text:
---
```

[3]:
```
'\ufeffThe Project Gutenberg EBook of The United States\' Constitution\rby
Founding Fathers\r\rCopyright laws are changing all over the world. Be sure to
check the\rcopyright laws for your country before downloading or
redistributing\rthis or any other Project Gutenberg eBook.\r\rThis header should
be the first thing seen when viewing this Project\rGutenberg file.  Please do
```

```
not remove it.  Do not change or edit the\rheader without written
permission.\r\rPlease read the "legal small print," and other information about
the\reBook and Project Gutenberg at the bottom of this file.  Included
is\rimportant information about your specific rights and restrictions in\rhow
the file may be used.  You can also find out about how to make a\rdonation to
Project Gutenberg, and how to get involved.\r\r\r**Welcome To The World of Free
Plain Vanilla Electronic Texts**\r\r**eBooks Readable By Both Humans and By
Computers, Since 1971**\r\r*****These eBooks Were Prepared By Thousands of
Volunteers!*****\r\r\rTitle: The United States\' Constitution\r\rAuthor:
Founding Fathers\r\rRelease Date: December, 1975  [EBook #5]\r[This file was
first posted on August 19, 2003]\r[Previous update: April 14, 2006]\r[Last
updated: April 1, 2015]\r\rEdition: 11\r\rLanguage: English\r\r\r*** START OF
THE PROJECT GUTENBERG EBOOK, THE UNITED STATES\' CONSTITUTION ***\r\r\r\r\rAll
of the original Project Gutenberg Etexts from the\r1970\'s were produced in ALL
CAPS, no lower case.  The\rcomputers we used then didn\'t have lower case at
all.\r\r***\r\rThese original Project Gutenberg Etexts will be compiled into a
file\rcontaining them all, in order to improve the content ratios of Etext\rto
header material.\r\r***\r\r\r\rThe following edition of The Consitution of the
United States of America\rhas been based on many hours of study of a variety of
editions, and will\rinclude certain variant spellings, punctuation, and
captialization as we\rhave been able to reasonable ascertain belonged to the
orginal.  In case\rof internal discrepancies in these matters, most or all have
been'
```

[4]:
```python
len(text.split(' '))
```

[4]: 6636

[5]:
```python
#https://stackoverflow.com/questions/46825980/huffman-coding-tree-traversal
#https://docs.python.org/3/library/heapq.html#module-heapq
#https://sites.fas.harvard.edu/~libs111/files/lectures/unit9-1.pdf
#https://www.cs.toronto.edu/~radford/csc310.F11/week3.pdf

heap_freq = []
for char, frq in freq.items():
    heap_freq.append([frq, [char, ""]])

#Sorts in decending order, places data in the heap structure.
heapfy = heapq.heapify(heap_freq)

#Loop through heap elements as it gets shorter.
while len(heap_freq) > 1:
    # Take the two smallest values from  the heap.
    left_node = heapq.heappop(heap_freq)
    right_node = heapq.heappop(heap_freq)

    # Return the list elements from left node.
```

```
    left_elements = left_node[1:]
    # Return the list elements from  the right node.
    right_elements = right_node[1:]

    # Scan each element in the left node and append a zero.
    for each in left_elements:
        each[1] = each[1] + '0'

    # Scan each element in the right node and append a one.
    for each in right_elements:
        each[1] = each[1] + '1'

    # Append the resulting elements (coded char and new freq) back into the heap.
    heapq.heappush(heap_freq, [left_node[0] + right_node[0]] + left_node[1:] + right_node[1:])

# Create dict of all of the coded char
huffDict = {each[0]:each[1] for each in heapq.heappop(heap_freq)[1:]}
huffDict
```

[5]:
```
{'s': '0000',
 'b': '001000',
 'y': '101000',
 'd': '11000',
 'e': '100',
 'i': '0010',
 'n': '1010',
 'k': '00000110',
 'N': '10000110',
 'T': '1000110',
 ',': '100110',
 ';': '000010110',
 'H': '100010110',
 ']': '00010010110',
 '7': '010010010110',
 '@': '0011001001O110',
 'X': '10110010010110',
 'Q': '1110010010110',
 '"': '1010010110',
 'G': '110010110',
 'S': '1010110',
 'v': '0110110',
 'O': '01110110',
 ':': '0011110110',
 'q': '1011110110',
 '*': '111110110',
 'a': '1110',
```

```
'l': '00001',
'm': '010001',
'I': '00110001',
'R': '10110001',
'A': '01110001',
"'": '00011110001',
'z': '10011110001',
'-': '1011110001',
'2': '0111110001',
'V': '1111110001',
'o': '1001',
't': '0101',
'w': '0001101',
'E': '01001101',
'O': '011001101',
'/': '00111001101',
'~': '000010111001101',
'\ufeff': '100010111001101',
'%': '10010111001101',
'8': '1010111001101',
'3': '110111001101',
'(': '01111001101',
')': '11111001101',
'D': '000101101',
'M': '100101101',
'L': '010101101',
'F': '0110101101',
'1': '1110101101',
'x': '001101101',
'Y': '0101101101',
'J': '1101101101',
'P': '11101101',
'h': '11101',
' ': '011',
'f': '000111',
'u': '100111',
'c': '010111',
'g': '0110111',
'.': '01110111',
'C': '11110111',
'r': '01111',
'p': '0011111',
'W': '0001011111',
'K': '001001011111',
'!': '101001011111',
'4': '011001011111',
'#': '0000111001011111',
```

```
 '$': '1000111001011111',
 '+': '0100111001011111',
 '<': '1100111001011111',
 '>': '0010111001011111',
 '_': '1010111001011111',
 '?': '110111001011111',
 '6': '1111001011111',
 'U': '101011111',
 'B': '011011111',
 'j': '0111011111',
 '9': '01111011111',
 '5': '011111011111',
 '[': '111111011111',
 '\r': '111111'}
```

[6]:
```
#Calculate Expected Information from the raw text.

df['entropy'] = -1 * df['freq'] * np.log2(df['freq'])
info_entropy = sum(df.entropy.to_list())
print('Information Entropy: {:.3f} bits'.format(info_entropy))
```

```
Information Entropy: 4.685 bits
```

[7]:
```
#Encode the raw text using Huffman Encoding based on starter code presented in␣
↪class.
#Calculated the Average Codeword Length.

encoded = ''
for i in range(len(text)):
    encoded += huffDict[text[i]]

print('Average Codeword Length After Encoding: {:.3f} bits'.format(len(encoded)/
↪len(text)))
```

```
Average Codeword Length After Encoding: 4.719 bits
```

[8]:
```
#Show the first 2000 characters of the encoded text.
print('The First 2,000 Characters of the Encoded Text: \n---\n{}'.
↪format(encoded[:2000]))
```

```
The First 2,000 Characters of the Encoded Text:
---
1000101110011011000110111011000111110110101111100101110111111000101110101011 1100
1011010011101011001010001000100011110110111011010011010110111111001100100000 1100
1110010001110111000110111011000111010111111010001001011001100001110101100101 1110
0101100000000001111000101111110111100110100000010100100101100111010100101001 10101
1111100100010100001101101101011011001100111101011000001010100110111011011010 1101
1110010111101100011110000111111111111111101111001001111110100001111001001101 1111
```

```
1010101011000011110000110100000111110011111000110101111110111101010011011100101
1001101110111110000010000101110010110110100011110110101111011000110001101100101
1100001110000111011101101101111110001100001001110111110001101011001011010111111
1100010111000001100110101111011001111110101111001001111110100001111001001101111
1010101011000011110000110100000110001111001011110111010001001100111011110110101
1100110011110100101011111010000110010001000001111001011111000111100010010001101
0100000110011110110000010101001101110111001011110110111110011000001000000101011
1001000100010011101010010101001101111111110101111010010000000111001011110111110
1010100001110010101111011000111101111110110101111100101110111111100010111010101
0010110100111010110010100010001000111101101110111000110111111001100100000110011
0111111111111111110001101110100100000001111101100111011000100011110110000111011
1001110000111000011001000100011010111101100011000111001001111000000101011010111
1001010100110111011000010010010100110001101111011001010011011011000101000001101
0101010011011101101011110100100000001111101101011111001011101111111000101110101
1111100101101001110101100101000100010001111011011101100011100100000110001110111
1101111110110100001100111000000100011110001001011101010010101011011111000100011
0110110100011001001010111011101101100010110110010111010100101010110101111110111
0101001101111000111001011110111001100000100101011010111101100111111111011001110
1000100011110110001101001001011110110011001110101011000110101111001001010101100
```

```
[9]: #Determine number of rows in Original DataFrame
print('Number of Rows in Original DataFrame: {}'.format(df.shape[0]))
```

```
Number of Rows in Original DataFrame: 88
```

```
[10]: #Determine number of rows in Encoded DataFrame
df_encode = pd.DataFrame({'char':  list(huffDict.keys()), 'encoded_char': ␣
↪list(huffDict.values())})
df_encode.shape
print('Number of Rows in Encoded DataFrame: {}'.format(df_encode.shape[0]))
```

```
Number of Rows in Encoded DataFrame: 88
```

```
[11]: #Show that all of rows have been maintained.
df = pd.merge(df, df_encode, on='char', how='inner').sort_values(by='freq',␣
↪ascending=False)
print('Number of Rows in Merged DataFrame: {}'.format(df.shape[0]))
```

```
Number of Rows in Merged DataFrame: 88
```

```
[12]: # Alternative calculation of the average codeword length
df['encoded_char_len'] = df['encoded_char'].str.len()
df['product'] = df['encoded_char_len'] * df['freq']
sum(df['product'].to_list())
print('Average Codeword Length After Encoding (method two): {:.3f} bits'.
↪format(sum(df['product'].to_list())))
```

```
Average Codeword Length After Encoding (method two): 4.719 bits
```

[13]:
```
#Return the 15 most frequent characters.
df.head(15)
```

[13]:

	char	char_count	freq	entropy	encoded_char	encoded_char_len
0		6635	0.155715	0.417786	011	3
1	e	4248	0.099695	0.331619	100	3
2	t	3094	0.072612	0.274738	0101	4
3	o	2568	0.060268	0.244233	1001	4
4	a	2358	0.055339	0.231072	1110	4
5	n	2273	0.053344	0.225568	1010	4
6	i	2114	0.049613	0.214979	0010	4
7	s	1962	0.046046	0.204479	0000	4
8	r	1927	0.045224	0.202005	01111	5
9	h	1586	0.037221	0.176717	11101	5
10	l	1267	0.029735	0.150806	00001	5
11	d	1079	0.025323	0.134297	11000	5
12	\r	994	0.023328	0.126479	111111	6
13	c	862	0.020230	0.113841	010111	6
14	u	839	0.019690	0.111572	100111	6

	product
0	0.467144
1	0.299085
2	0.290448
3	0.241070
4	0.221356
5	0.213377
6	0.198451
7	0.184182
8	0.226121
9	0.186107
10	0.148674
11	0.126613
12	0.139967
13	0.121380
14	0.118141

[14]:
```
#Return the 15 least frequent characters.
df.tail(15)
```

[14]:

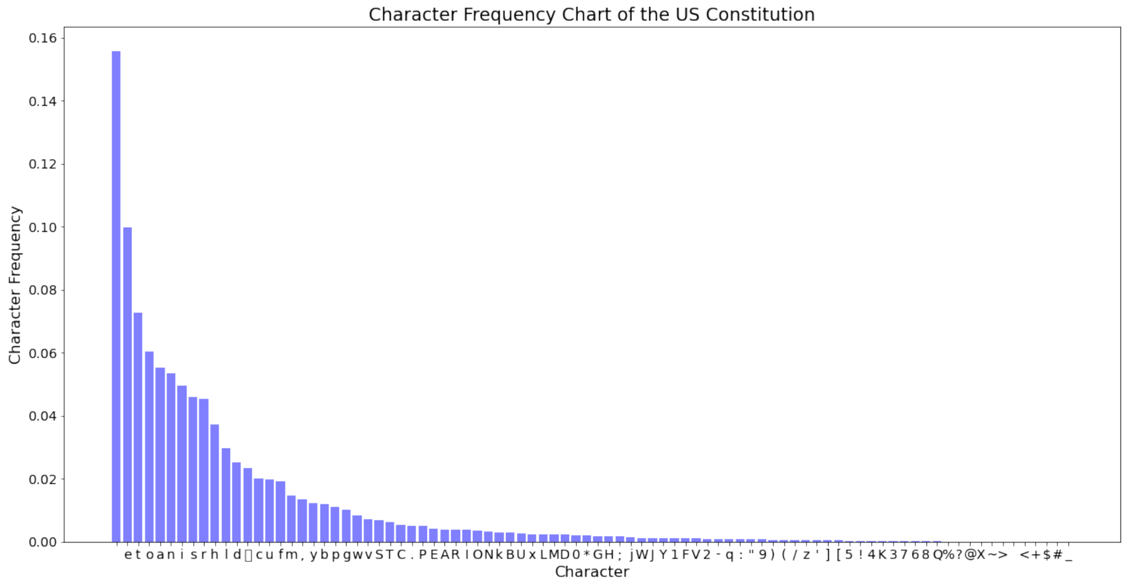
	char	char_count	freq	entropy	encoded_char	encoded_char_len
73	6	8	0.000188	0.002324	1111001011111	13
74	8	6	0.000141	0.001802	1010111001101	13
75	Q	5	0.000117	0.001532	1110010010110	13
76	%	3	0.000070	0.000971	10010111001101	14
79	?	2	0.000047	0.000675	110111001011111	15
77	@	2	0.000047	0.000675	00110010010110	14

```
78     X           2  0.000047  0.000675    10110010010110                 14
80     ~           1  0.000023  0.000361   000010111001101                 15
81     >           1  0.000023  0.000361  0010111001011111                 16
82               1  0.000023  0.000361   100010111001101                 15
83     <           1  0.000023  0.000361  1100111001011111                 16
84     +           1  0.000023  0.000361  0100111001011111                 16
85     $           1  0.000023  0.000361  1000111001011111                 16
86     #           1  0.000023  0.000361  0000111001011111                 16
87     _           1  0.000023  0.000361  1010111001011111                 16

     product
73  0.002441
74  0.001831
75  0.001525
76  0.000986
79  0.000704
77  0.000657
78  0.000657
80  0.000352
81  0.000375
82  0.000352
83  0.000375
84  0.000375
85  0.000375
86  0.000375
87  0.000375
```

[15]:

```python
#Plot the distribution of char from the raw text.

plt.figure(figsize=(20, 10))
plt.bar(df.char, df.freq, color='blue', alpha=0.5)
plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.xlabel('Character', fontsize=16)
plt.ylabel('Character Frequency', fontsize=16)
plt.title('Character Frequency Chart of the US Constitution', fontsize=19)
plt.show()
```

```
/opt/miniconda3/envs/ssie/lib/python3.8/site-
packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 13
missing from current font.
  font.set_text(s, 0.0, flags=flags)
/opt/miniconda3/envs/ssie/lib/python3.8/site-
packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 13
missing from current font.
  font.set_text(s, 0, flags=flags)
```


Character Frequency Chart of the US Constitution
Character Frequency
Character
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00