Information Theory
Project 2

Introduction

The following project consists of the task of compressing in binary form a file using optimal coding techniques. In particular the coding is generated by dividing the file in ascii characters and generating codes for them using Huffman coding.

The code is generated by using the frequency distribution of these characters with the objective of assigning shorter codes to more frequent characters and thus reducing the average code length and the size of the encoded file.

It is important to note that Huffman coding ensures that the code will be prefix, this means that for no two characters the code of one is a prefix of the code of the other. This property makes it possible to decode the file back to its original form without loss of information.

The project is implemented in python and a binary tree is used in order to generate the codes, as explained in the lectures. The script and classes used can be found in the appendix.

Character Distribution

We start by counting the number of times each character appears in the text. To get optimal coding we expect that the higher the count the shorter the code. We divide the count of each character by the total number of characters to get the distributions. In this particular case the total number of characters is 148,481 as counting took place in a Linux system.

Char	Probability	Char	Probability	Char	Probability	Char	Probability
------	-------------	------	-------------	------	-------------	------	-------------
‘\x1a’	6.73487e-06	Q	0.000565729	:	0.00156922	c	0.0151737
2	6.73487e-06	B	0.000612873	W	0.00159616	,	0.0162849
9	6.73487e-06	L	0.000660017	H	0.0019127	w	0.0164129
Z	6.73487e-06	"	0.00076104	!	0.00302396	g	0.0164735
[	1.34697e-05	Y	0.000767775	T	0.00317886	u	0.022912
]	1.34697e-05	N	0.000808184	A	0.00429685	‘\n’	0.0242994
X	2.69395e-05	q	0.000841859	-	0.00450563	l	0.0310814
_	2.69395e-05	j	0.000929412	I	0.00493666	d	0.0319165
J	5.38789e-05	R	0.000942882	v	0.0054081	r	0.0356477
V	0.000282864	C	0.000969821	.	0.00657997	s	0.0422748

)	0.000370418	x	0.000969821	k	0.00724672	i	0.0456489
(	0.000377153	O	0.00118534	`	0.00746223	n	0.0464234
*	0.000404092	E	0.00126616	b	0.00931432	h	0.0477367
P	0.000431032	D	0.00129309	p	0.00981944	o	0.0536432
U	0.000444501	;	0.00130656	'	0.0118601	a	0.0548824
F	0.00049838	M	0.00134697	m	0.0128434	t	0.0687765
z	0.000518585	?	0.00136044	f	0.0129714	e	0.0901193
G	0.000552259	S	0.0014682	y	0.01448	‘ ‘	0.194638
K	0.000552259						

Code and Stats

Now that we have the distribution of the characters we can proceed to generate their encoding. This is done by building a tree which contains the characters as leaves and which nodes are ordered based on their probabilities. This algorithm was presented in the lectures. The implementation can be found in the appendix.

The basic idea is that each step that you make starting from the root either adds a 0 or a 1 to the code, depending if you move to the right or the left. The characters which have higher probabilities will be closer to the root, thus having shorter codes. Given that characters can only be found in the leaves, it can not be the case that one code is a prefix of another which implies the prefix property.

Char	Code	Char	Code	Char	Code	Char	Code
------	-------------	------	-------------	------	-------------	------	-------------
‘\x1a’	1000010011000101	Q	11001011000	:	010111000	c	100011
2	1000010011000111	B	11001011001	W	010111001	,	101001
9	1000010011000100	L	11110100110	H	100001011	w	101010
Z	1000010011000110	"	11110101010	!	111101011	g	101011
[	100001001100001	Y	11110101011	T	01011101	u	111100
]	100001001100000	N	1000010000	A	10100011	‘\n’	01010
X	100001001100110	q	1000010010	-	11001010	l	10010
_	100001001100111	j	1000010100	I	11001110	d	10011
J	10000100110010	R	1000010101	v	11001111	r	11000
V	100001001101	C	1010001001	.	0101111	s	11010
)	111101001110	x	1010001000	k	1000011	i	11011
(	111101001111	O	1100101101	`	1010000	n	11111
*	10000100010	E	1100101110	b	1100100	h	0100
P	10000100011	D	1100101111	p	1100110	o	0110
U	10000100111	;	1111010000	'	1111011	a	0111
F	10100010100	M	1111010001	m	010110	t	1011
z	10100010101	?	1111010010	f	100000	e	1110

G K	10100010111 10100010110	S	1111010100	y	100010	‘ ‘	00

With the codes for the characters we can get some numeric values to better understand the problem and how efficient the compression is. Let X be the random variable which assigns to each character its generated code and assume that in the original file each character is represented via its ascii value consisting of 8 bits (thus 8 bit average word length). The entropy is calculated using the methods implemented for the first project and the average word length consists of simply weighting the lengths in bits of the codes with the distribution presented in the last section.

Entropy (H(X))	4.512876838738921
Average word length (L)	4.555289902411756 b
Size of original file	1187848 b
Size of compressed file	676374 b

As expected, the average word length is bounded from below by the entropy and the size in bits of the compressed file is much smaller than the size of the original file.

It is interesting to see that there are codes which are much bigger than 8 bits, for example the code assigned to the character ‘2’. Even then the size of the compressed file is about half of the size of the original file. This is because the code is optimized for characters with high probabilities as ‘ ‘ or ‘e’ which have a 2 bit code and a 4 bit code respectively.

Appendix

Script

```
import sys
sys.path.insert(0,"..")
import numpy as np
from Project1.HandIn1 import InfoTheory
from Project2.huffman_binary_tree import HuffmanBinaryTree
from tabulate import tabulate

with open('Alice29.txt', 'r') as file:
   rawChars = np.array([char for line in file for char in line])

chars, counts = np.unique(rawChars, return_counts=True)
char_count_dict = dict(zip(chars, counts))
```

```
totalChars = rawChars.size
charDistr = counts/totalChars

char_prob_dict = dict(zip(chars, charDistr))
sorted_char_prob_dict = dict(sorted(char_prob_dict.items(), key=lambda
item: item[1]))
print(tabulate(sorted_char_prob_dict.items(), headers=['Char',
'Probability']))

hbt = HuffmanBinaryTree(sorted_char_prob_dict)
code_dict = hbt.get_codes()
print(tabulate(code_dict.items(), headers=['Char', 'Code']))

IT = InfoTheory()
entropy = IT.Entropy(np.array([charDistr]))
print(f"Entropy: {entropy[0]}")

avg = 0
for char in chars:
   avg += len(code_dict[char])*sorted_char_prob_dict[char]
print(f"Average word length: {avg}")

compressed_len = 0
for char in chars:
   compressed_len += len(code_dict[char])*char_count_dict[char]
print(f"Size of original file: {rawChars.size*8}")
print(f"Size of compressed file in bits: {compressed_len}")
```

Huffman Code Tree Implementation

```
import numpy as np
import heapq as hq

class HuffmanBinaryTree:

   def __init__(self, word_prob_dict):
       self.word_proc_dict = word_prob_dict
       self.leaves = {key: self.HuffmanBinaryNode(value, key) for key,
value in word_prob_dict.items()}

       self.root = self.build_tree_from_leaves()
```

```python
    self.generate_codes(self.root, "")

class HuffmanBinaryNode:

    def __init__(self, probability, word=None):
        self.probability = probability
        self.word = word
        self.right = None
        self.left = None
        self.code = None

    #We order the nodes of the tree based on their probability
    def __lt__(self, other):
        return self.probability < other.get_probability()

    def is_leaf(self):
        return self.right is None and self.left is None

    def get_probability(self):
        return self.probability

    def set_right(self, right):
        self.right = right

    def set_left(self, left):
        self.left = left

    def get_right(self):
        return self.right

    def get_left(self):
        return self.left

    def set_code(self, code):
        self.code = code

    def get_code(self):
        return self.code

def build_tree_from_leaves(self):
    node_heap = list(self.leaves.values())
    hq.heapify(node_heap)
```

```python
        min_node = hq.heappop(node_heap)
        while len(node_heap) > 0:
            next_min_node = hq.heappop(node_heap)
            new_node = self.HuffmanBinaryNode(min_node.get_probability()
+ next_min_node.get_probability())

            new_node.set_right(min_node)
            new_node.set_left(next_min_node)

            hq.heappush(node_heap, new_node)
            min_node = hq.heappop(node_heap)

        return min_node

    #The algorithm for generating codes over the tree will be
implemented recursively
    #This is because the number of words encoded is small
    def generate_codes(self, root, code):

        if root.is_leaf():
            root.set_code(code)
            return

        #Both children should exist as there can be no unused leaves
        self.generate_codes(root.get_right(), code + "0")
        self.generate_codes(root.get_left(), code + "1")

    def get_codes(self):
        return {key: value.get_code() for key, value in
self.leaves.items()}
```