Lund University, Information Theory

First hand in problem

Sergio Gutierrez,

May 2019

Exercise

In this problem a text should be compressed using Huffman coding. The text is a well used reference text in source coding, Alice's adventures in wonderland by Lewis Carroll. The version used here is from http://corpus.canterbury.ac.nz/ as Alice29.txt, in the Canterbury Corpus. It is also given with the same name at the course instant at Moodle (elearning.eit.lth.se). In the problem you first need to estimate the probability function for the letters included in the file, and then construct an optimal source code based on this estimation. Use the code to compress the text, and compare the average codeword length with the entropy of the estimated distribution.

Solution

For this problem we will use a computational solution done in Python and following Huffman coding algorithm. For sake of simplicity we will assume that the reader has a working knowledge of the Huffman algorithm and we will directly approach the task at hand going over some few interesting points that showed up during the computational solution. The complete code can be seen at the end of the text.

Now the first task is to load the file and compare the number of characters found in the text including any end line or symbols of the same nature. One can approach this in many forms we looked at two different scheme or modes since we found an interesting change between both of this loading schemes. In python we used the command open(filename, mode) and tested two mode 'r' and 'rb' or read and read by byte with these two mode we got a different amount of X elements. In the case of read we detected 148481 and for read by byte a total

of 152089 elements were found, one would not comment this if there were no difference but there is a slight deviation that is commented later. Let us now look at the Huffman code it self.

The approach taken was first constructing a set of list recursively that contained all the information of the node trees and the prefix. At the end of a single cycle we remove the two lowest probability element and replace it for the node of the previous leaf. With this vectors constructed we can proceed to assign the prefix using the node array, since this array contains all the nodes and leafs we can just iterate over the different element names and recursively set the prefixes.

Consider the first case we have a sorted vector s_vector that only has two elements let say 'a' and 'ftye' our function assign_name() will call for

$$node['aftye'] = ['a', 'ftye']$$

its clear that the length of this element is two, so our function calls its self again and iterates over prefix for each root by

$$[prefix +' 0', prefix +' 1']$$

. When the function is called for the first element

$$node['a'] = []$$

which is length zero the function returns the prefix 0 since prefix=" at the start. At the same time for the second leaf we have

$$node['ftye'] = ['f', 'tye']$$

and for prefix we have

$$['1' +' 0', '1' +' 1']$$

, the function is called again

$$node['f'] = []$$

giving prefix '10' for the other leaf

$$node['tye'] = ['t', 'ye']$$

with

$$['11' +' 0', '11' +' 1']$$

this will continue until all the tree is covered giving off all the prefixes.

The encoded file can also be decoded following the same logic of searching threw the tree but it wont be discussed in detail since it was not necessary for the solution. The difference in mode when using open file comes into play here

when using read mode we get a less elements than expected but at the time of decoding the encoded file we get that the decode file is the same size as the original file 149 KB. When using read by byte we get the correct number of elements in the text but the decoded file is 153 KB which is slightly larger than the original file. In the read by byte we are able to find one more element but at the time of comparing the files between the two modes they look the same with out the extra symbol. I have attached only the table with all the elements but the code has both decompressing methods.

Finally we look at the average code word length and the Entropy which can be computed directly from the file called Sol.txt that comes out as output from the total program. The average code word length was 4.61 and the entropy 4.58 which is consistent with the theoretical model.

data.txt

X	Freq	Prob	Code Word
' '	28900	1.9002e-01	00
'e'	13381	8.7981e-02	1101
't'	10212	6.7145e-02	1011
'a'	8149	5.3580e-02	0111
'o'	7965	5.2371e-02	0110
'h'	7088	4.6604e-02	11111
'n'	6893	4.5322e-02	11110
'i'	6778	4.4566e-02	11100
's'	6277	4.1272e-02	11001
'r'	5293	3.4802e-02	10101
'd'	4739	3.1159e-02	10010
'l'	4615	3.0344e-02	10001
'\n'	3608	2.3723e-02	01000
'\r'	3608	2.3723e-02	01001
'u'	3402	2.2368e-02	111010
'g'	2446	1.6083e-02	101001
'w'	2437	1.6024e-02	101000
','	2418	1.5899e-02	100111
'c'	2253	1.4814e-02	100001
'y'	2150	1.4136e-02	100000
'f'	1926	1.2664e-02	010110
'm'	1907	1.2539e-02	010100
"'"	1761	1.1579e-02	1110111
'p'	1458	9.5865e-03	1100010
'b'	1383	9.0934e-03	1100000
'‘'	1108	7.2852e-03	1001100
'k'	1076	7.0748e-03	0101111
'.'	977	6.4239e-03	0101011
'v'	803	5.2798e-03	11000111
'I'	733	4.8195e-03	11000110
'-'	669	4.3987e-03	11000010
'A'	638	4.1949e-03	10011011
'T'	472	3.1034e-03	01010101
'!'	449	2.9522e-03	111011011
'H'	284	1.8673e-03	010111011
'W'	237	1.5583e-03	010101001
':'	233	1.5320e-03	010101000

'S'	218	1.4334e-03	1110110100
'?'	202	1.3282e-03	1110110010
'M'	200	1.3150e-03	1110110001
';'	194	1.2756e-03	1110110000
'D'	192	1.2624e-03	1100001111
'E'	188	1.2361e-03	1100001110
'O'	176	1.1572e-03	1100001101
'C'	144	9.4681e-04	1001101001
'x'	144	9.4681e-04	1001101000
'R'	140	9.2051e-04	0101110101
'j'	138	9.0736e-04	0101110100
'q'	125	8.2189e-04	0101110010
'N'	120	7.8901e-04	0101110000
'Y'	114	7.4956e-04	11101101011
'"'	113	7.4299e-04	11101101010
'L'	98	6.4436e-04	11101100110
'B'	91	5.9833e-04	11000011001
'Q'	84	5.5231e-04	11000011000
'G'	82	5.3916e-04	10011010111
'K'	82	5.3916e-04	10011010110
'z'	77	5.0628e-04	10011010101
'F'	74	4.8656e-04	10011010100
'U'	66	4.3396e-04	01011100110
'P'	64	4.2081e-04	01011100011
'*'	60	3.9451e-04	01011100010
'('	56	3.6821e-04	111011001111
')'	55	3.6163e-04	111011001110
'V'	42	2.7615e-04	010111001111
'J'	8	5.2601e-05	0101110011100
'X'	4	2.6300e-05	0101110011110100
'_'	4	2.6300e-05	0101110011110111
'['	2	1.3150e-05	0101110011101011
']'	2	1.3150e-05	0101110011101101
'\x1a'	1	6.5751e-06	01011100111011000
'2'	1	6.5751e-06	01011100111010100
'9'	1	6.5751e-06	01011100111011001
'Z'	1	6.5751e-06	01011100111010101

Huffman Code

May 9, 2019

```
In [1]: %pylab inline
        %matplotlib inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

1 Huffman Coding

1.1 Reading file

1.1.1 Load file approche 1

```
In [2]: x=open('alice29.txt')
        book=[]
        for line in x:
            book.append(line)
        x.close()
```

```
In [3]: char=[]
        for i in range(size(book)):
            if(size(list(book[i]))>1):
                char.append(list(book[i]))
            else:
                char.append(list(book[i]))
```

```
In [4]: Xelm=[]
        text= ''
        elem = ''
        for i in range(size(char)):
            for j in range(size(char[i])):
                Xelm.append(char[i][j])
                text = text+char[i][j]
```

1.1.2 Load file approche 2

```
In [5]: with open('alice29.txt', 'rb') as inf:
            book = inf.read()
```

```
In [6]: text=book.decode('utf-8')
        Xelm=list(text)
```

1.2 Constructing the probability distribution

```
In [7]: logi= True
        uniq = []
        n=0
        for i in range(len(Xelm)):
            if(i==0):
                uniq.append(Xelm[i])
            else:
                for j in range(len(uniq)):
                    if(Xelm[i]!=uniq[j]):
                        n=1
                    else:
                        logi = False

            if(logi == True and i!=0):
                uniq.append(Xelm[i])
            logi= True
        uniq.sort()
```

```
In [8]: n=0
        count = []
        for i in range(len(uniq)):
            for j in range(len(Xelm)):
                if(uniq[i]== Xelm[j]):
                    n=n+1
            count.append(n)
            n=0
```

```
In [9]: dist=[]
        for i in range(len(uniq)):
            dist.append((count[i],uniq[i]))
```

```
In [10]: dist.sort(key=lambda x:x[0],reverse=True)
```

1.3 Huffman code

```
In [14]: def assign_name(nodes, root, result, prefix = ''):
             leaf = nodes[root]
             tree = {}
             if len(leaf) == 2:
                 tree['0'] = assign_name(nodes, leaf[0], result, prefix+'0')
                 tree['1'] = assign_name(nodes, leaf[1], result, prefix+'1')
                 return tree
             else:
                 result[root] = prefix
                 return root
```

```
In [15]: def Huffman_code(_vals):
             vals = _vals.copy()
```

```
    nodes = {}
    for n in vals.keys(): # leafs initialization
        nodes[n] = []

    while len(vals) > 1: # binary tree creation
        s_vals = sorted(vals.items(), key=lambda x:x[1])
        p1 = s_vals[0][0]
        p2 = s_vals[1][0]
        vals[p1+p2] = vals.pop(p1) + vals.pop(p2)
        nodes[p1+p2] = [p1, p2]
    out = {}
    root = p1+p2
    tree = {}
    tree = assign_name(nodes, root, out)    # assignment of the code for th
    return out, tree, nodes[p1]
```

Creating tree for our file

```
In [16]: freq=dist
         vals = {l:v for (v,l) in freq}
         code, tree,nodes = Huffman_code(vals)

In [17]: encoded = ''.join([code[t] for t in text])

         decoded = []
         i = 0
         while i < len(encoded):
             ch = encoded[i]
             act = tree[ch]
             while not isinstance(act, str):
                 i += 1
                 ch = encoded[i]
                 act = act[ch]
             decoded.append(act)
             i += 1
```

Solution output

```
In [18]: sol=[]
         for i in range(len(dist)):
             sol.append((dist[i][1],dist[i][0],dist[i][0]/sum(count), code[dist[i]|

In [19]: def wrtsol(arr,name):
             f= open("%s"%name,"+w")
             f.write("      X      Freq       Prob       Code Word \n")
             for  i in range(len(arr)):
                 f.write("     %r     %i     %.4e     %s \n" %(arr[i][0],arr[i][1],
             f.close()
```

```
        def wrt(arr,name):
            f= open("%s"%name,"+w")
            arr=decoded
            for  i in range(len(arr)):
                f.write("%s" %arr[i])
            f.close()

In [20]: wrtsol(sol,'sol1.txt')
         wrt(decoded,'decoded.txt')
```

1.4 Code word length and Entropy

```
In [21]: lx=[]
         for i in range(len(sol)):
             lx.append(sol[i][2]*len(list(sol[i][3])))
         sum(lx)

Out[21]: 4.612444029482738

In [22]: E=[]
         for i in range(len(sol)):
             E.append(sol[i][2]*log2(sol[i][2]))
         -sum(E)

Out[22]: 4.5676802121772644
```