Glasgow College, UESTC

Elements of Information Theory

Source Coding for *the Game of Thrones*

Author: Changgang Zheng & Mengyu Ge
UoG ID: 2289258Z & 2289225G
UESTC ID: 2016200302027 & 2016200301029
E-mail: 2016200302027@std.uestc.edu.cn & 2016200301029@std.uestc.edu.cn

University of Glasgow

电子科技大学 University of Electronic Science and Technology of China

SOURCE CODING FOR THE GAME OF THRONES

GEMENG YU AND CHANGGANG ZHENG[1]

November 26, 2018

CONTENTS

ABSTRACT

This report mainly introduces several common coding methods and uses *Game of Thrones* as the source file for encoding and decoding. For these coding methods, we use different programming languages to implement them, and finally render them as executable files.

1*Glasgow College, University of Electronic Science and Technology of China, ChengDu, China*

1 INTRODUCTION

Coding theory is a branch of mathematics and computer science dealing with the error tendency in transmitting data in noisy channels. According to coding theory, better methods are used for data transmission to correct a large number of errors during transmission. There are two types of coding: source coding (data compression [also known as data compression]) and channel coding (forward error correction). This report focuses on the former one.

2 HUFFMAN CODING

Huffman coding algorithm is based on binary tree to build coding compression structure, which is a classic algorithm in data compression. The algorithm reencodes the characters according to the frequency of text characters. To shorten the length of the code, we naturally want words with higher frequency to encode shorter, so as to finally maximize the compression of the space to store text data.

Through the study of Huffman coding, we find that it has both advantages and disadvantages. The advantage is that it is a distortion-free data compression coding, that means the original data can be restored without distortion after decoding. The second advantage is that Hofmann code is the shortest code length which saving space effectively. However, there are also several disadvantages. It can only be expressed as integers rather than as decimals, which greatly limits compression. Besides, All of Hoffman's bits are grouped together, and changing one of them can make the data look completely different. What is more, only when the probability of each symbol of the information source is extraordinarily uneven can the Huffman coding effect be obvious. Huffman coding must be able to accurately count the frequency of each symbol in the original file, without which the compression effect cannot achieve our expected. Huffman coding also usually goes through two operations, the first for statistics, and the second for producing code, so the coding efficiency is relatively low.

Before defining the Huffman tree. Here we explain a few concepts related to the Huffman tree.

- Path:A branch between one node in the tree and another node constitutes a path between these two nodes.
- Path length:The number of branches on a path.
- The path length of the tree:The length of the path from the root to each node.
- Weight Path Length:In a tree, if a node is attached with a weight, the path length of the node is usually multiplied by the weight of the node.

- Weighted Path Length of Tree(WPL):If each leaf in the tree has a weight, the sum of all leaf weight path lengths in the tree is called the tree weight path length.

Table 1: Huffman coding

Character	Frequency	Encoding	Character	Frequency	Encoding
P	759	10111010100	T	6094	10111001
R	2096	000100101	i	69083	0000
O	934	10111011101	D	1794	1011101100
L	2780	10111011101	f	25057	101100
G	1005	10111011101	y	25269	101101
U	186	10111011101	S	4774	00010110
E	978	10111011110	j	762	10111010101
W	2436	000100111	H	3974	00010001
e	157340	010	p	13911	1100110
blank	281965	111	v	8828	0001010
s	78177	0010	A	3000	101110100
h	85523	0111	I	5104	00010111
o	94190	1000	q	978	10111011111
u	30489	110100	M	1959	000100000
l	52851	11000	N	1982	000100001
d	66066	11011	Y	1404	1011100010
t	98608	1010	B	1998	000100100
a	97696	1001	x	605	00010011010
r	78829	0011	F	814	10111010110
b	18740	000110	z	609	00010011011
c	21754	000111	C	1455	1011100011
k	14042	1100111	J	1832	1011101101
g	26469	101111	K	903	10111011100
w	30936	110101	V	512	101110101111
n	81760	0110	Q	128	10111010111001
m	27948	110010	X	6	10111010111000

If a binary tree has n weighted leaf nodes, the length of the weighted path is denoted as:

$$WPL = \sum_{k=1}^{n} w_k l_k$$

Where W_k is the weight of the leaf node; L_k is the path length of this node.

According to the definition of Huffman tree. To minimize its WPL value of a binary tree, the higher the weight, the closer to the root, and the smaller the weight, the farther from the root. So, the steps for the Huffman tree are as follows:

1 For given n weights noted as $w_1, w_2, \ldots, w_3$. Constructing n binary trees with only root nodes and set their weight as w_j.

2 In the forest, two trees with the minimum weight of root node are selected as the left and right subtrees. Then a new binary tree is constructed, and the weight of the new binary tree root node is set as the sum of the weight of the left and right child nodes.

3 Remove two trees used in last step from the forest and add the new binary tree to the forest.

4 Repeat steps 2 and 3 until there is only one tree, and this is the Huffman tree.

In message transmission, the binary encoding of each character that appears in the message is required to follow two principles in the design of the code. Firstly, the binary code transmitted by the sender must be unique to the receiver after decoding, that is, the decoding result must exactly the same as the message sent by the sender. Secondly, the binary encoding sent should be as short as possible. In the "game of thrones" document presented, we code based on the following two points:

1 A Huffman tree is constructed using the frequency of each character in the character set as the weight.

2 Starting from the root, assign 0 to the left branch and 1 to the right branch, and from the root to the leaf direction to form the leaf node coding.

Through this experiment, we realized the Huffman compression code of "game of thrones". The main difficulty we encountered was constructing Huffman trees and dictionaries from input data. The main idea is to read the characters one by one from the input data then record the number of occurrences and the total number of characters in the file to determine frequency. Then we create the Huffman tree based on the character frequency and find the Huffman code. Finally, we output it.

3 SHANNON CODING

Shannon coding theorem is also an common theorem. It is an variable length coding. In this method, code length is depend on the cumulative probability. The code word is depend on its own probability. Technically speaking, Shannon coding is not the best coding method, it just assign the code by using the cumulative probability.

For Shannon coding, there are advantages as follow. The first is that, we do not need to care about the dummy leave in order to meet the formula D+D(K-1).

The second is that it is easier to calculate and implement. The disadvantage also exists as it may not be the beast coding method.

Shannon coding is not the optimal code in the strict sense. It uses the cumulative probability distribution function of the source symbol to allocate the code word. The implementation of Shannon coding is as follows:

Table 2: Shannon coding

Character	Frequency	Encoding	Character	Frequency	Encoding
blank	281965	000	H	3974	111110100
e	157340	0010	A	3000	1111101010
t	98608	0100	L	2780	1111101100
a	97696	0101	W	2436	1111101110
o	94190	01101	R	2096	1111110000
h	85523	01111	B	1998	1111110001
n	81760	10000	N	1982	1111110010
r	78829	10010	M	1959	1111110100
s	78177	10100	J	1832	1111110101
i	69083	10101	D	1794	1111110110
d	66066	10111	C	1455	11111101111
l	52851	11000	Y	1404	11111110001
w	30936	110011	G	1005	11111110011
u	30489	110100	E	978	11111110101
m	27948	110110	q	978	11111110110
g	26469	110111	O	934	11111110111
y	25269	111000	K	903	11111111000
f	25057	111001	F	814	11111111010
c	21754	1110101	j	762	11111111011
b	18740	1110111	P	759	11111111100
k	14042	1111000	z	609	111111111010
p	13911	1111001	x	605	111111111100
v	8828	11110101	V	512	111111111101
T	6094	11110111	U	186	11111111111100
I	5104	111110000	Q	128	11111111111110
S	4774	111110010	X	6	111111111111111110

1 Sort the probability in descending order.

$$P(a_1) \geqslant P(a_2) \geqslant ... \geqslant P(a_q)$$

2 Get the cumulative probability in the same order and make sure the integer length satisfies the following inequality.

$$-\log_2 p(x_i) \leqslant K_i \leqslant -\log_2 p(x_i) + 1$$

3 Calculate the logD(prob) as the code length.

$$P_i = \sum_{k=1} i - 1 P(a_k)$$

4 Calculate the binary of the each cumulative probability and cut the corresponding length to form the code.

In this way, Shannon coding is finished. By using this method, we can find that the coding length is get by calculate the logD(cumulative). The more the probability is, the shorter code length it will be. This can guarantee the efficiency. When we use the binary code, the coding efficacy and reach 100% if the each probability are Negative powers of two. More tricky, the binary of each cumulative probability forms a prefix-free code, the code word can be decoded with no ambiguity. We can also know that the efficiency of Shannon coding is not very high, practicality is also not perform well, but there has a good theoretical guidance meaning for other encoding method. In general,the average code length of codes compiled by shannon coding method is not the shortest (optimal code). Only when the probability distribution of the source symbol makes the equal sign on the left side of the inequality true can the coding efficiency reach the highest

4 FANON CODING

This method dates from the year 1949. It was published by Claude Elwood Shannon (he is designated as the father of theory of information) with Warren Weaver and by Robert Mario Fano independently. Similar to huffman-tree, fano coding also uses a binary Tree to encode characters. However, in practice, fano coding is not of great use. This is because it has lower coding efficiency compared with Huffman coding (or the average code word of fano algorithm is larger), but its basic ideas can be referred.

Fanon coding algorithm is based on dividing nodes by even probability, it is a classic algorithm in data compression. The algorithm reencodes the characters according to the frequency of text characters. To shorten the length of the code, we naturally want words with higher frequency to encode shorter, so as to finally maximize the compression of the space to store text data.

Compare with the Huffman codes, the Fanon coding also tends to construct a code tree. Different from Huffman codes, this method construct the tree from the top to bottom. The idea are simple and the main steps are as follow.

1 For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol's relative frequency of occurrence is known.

2 Sort the lists of symbols according to frequency, with the most frequently occurring symbols at the bottom and the least common at the top.

3 Divide the list into two parts, with the total frequency counts of the left part being as close to the total of the right as possible.

4 The left part of the list is assigned the binary digit 0, and the right part is assigned the digit 1. This means that the codes for the symbols in the first part will all start with 0, and the codes in the second part will all start with 1.

5 Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

Pseudo-code

```
1: begin
2:   count source units
3:   sort source units to non-decreasing order
4:   SF-SplitS
5:   output(count of symbols, encoded tree, symbols)
6:   write output
7:  end
8:
9: procedure SF-Split(S)
10: begin
11:   if (|S|>1) then
12:    begin
13:      divide S to S1 and S2 with about same count of units
14:      add 1 to codes in S1
15:      add 0 to codes in S2
16:      SF-Split(S1)
17:      SF-Split(S2)
18:    end
19: end
```

As it can be seen in pseudocode of this algorithm, there are two passes through an input data. Based on frequency of appearance of symbols in the input text, codes for each symbol are generated. Using a method from top to bottom, the prefix code of variable length is generated. First, symbols are ordered by counts of their appearance. Next, list of symbols is recursively divided into two parts of almost same frequency of symbols occurrence.

First, count of different symbols is written to the output. Next, the encoded tree is written to the output. This tree is constructed from parts of divided list of symbols. Left branches of the tree are marked by 1, right branches by 0. This

numbers are created by walk through the tree from root and from left to right. Only necessary numbers are written. Next, Fibonacci code of count of leaves is determined and written to the output. Then the coding of the input begins.

Table 3: Fanon coding

Character	Frequency	Encoding	Character	Frequency	Encoding
blank	281965	000	H	3974	111110101
e	157340	001	A	3000	111110110
t	98608	010	L	2780	111110111
a	97696	0110	W	2436	111111000
o	94190	0111	R	2096	1111110010
h	85523	1000	B	1998	1111110011
n	81760	1001	N	1982	111111010
r	78829	1010	M	1959	1111110110
s	78177	10110	J	1832	1111110111
i	69083	10111	D	1794	1111111000
d	66066	11000	C	1455	1111111001
l	52851	11001	Y	1404	1111111010
w	30936	11010	G	1005	11111110110
u	30489	110110	E	978	11111110111
m	27948	110111	q	978	11111111000
g	26469	111000	O	934	11111111001
y	25269	111001	K	903	11111111010
f	25057	111010	F	814	11111111011
c	21754	111011	j	762	11111111100
b	18740	1111000	P	759	11111111101
k	14042	1111001	z	609	111111111100
p	13911	1111010	x	605	111111111101
v	8828	1111011	V	512	111111111110
T	6094	11111000	U	186	1111111111110
I	5104	11111001	Q	128	11111111111110
S	4774	111110100	X	6	11111111111111

5 SUMMARIZE ANALYSIS AND EXPERIENCE

We first do the preprossing to the raw text and based on the aboving computing and analysis. We apply the encoing and decoding to the source code of the txt: 'the Game of Thrones'. The coding process is successful and we can get the following result, as show in Table 4. From the table, we can see that the Huffman coding has the highest coing effiency and the Shannon coding have the worst. The Fano coding's effiency is in the middle place.

Table 4: Coding efficiency comparison

	Huffman	Shannon	Fano
Coding efficiency	1	89.286	98.336
Entropy	4.2668	4.2668	4.2668
Average length	4.3042	4.7788	4.3390

6 SOME OTHER CODING METHOD

6.1 Arithmetic Coding

Arithmetic coding is a lossless data compression method and an entropy coding method. Different from other entropy coding methods, which usually divide the input message into symbols and then encode each symbol. Arithmetic coding is the direct encoding of the entire input message into a number which between 0 and 1. Arithmetic coding uses two basic parameters: the probability of the symbol and its coding interval. The probability of the source symbol determines the efficiency of the compression encoding and the spacing of the source symbols during the encoding process, which is contained between 0 and 1. The algorithm idea of arithmetic coding is as follows

1 For a group of source symbols, the probability of the symbol is sorted from large to small, [0,1] is set as the current analysis interval and the proportional interval is divided according to the probability sequence of source symbols in the current analysis interval.

2 Retrieving the input message sequence, lock the current message signs first retrieve the word is the first message (symbol) to find the current interval interval, the proportion of the symbol in the current analysis to this interval as a new starting point of the current analysis and analysis of the current interval (namely the left endpoint) indicating the number of adding to encode output current message symbols in the number of Pointers.

3 The proportional interval is still divided according to the probability sequence of the source symbol in the current analysis interval.Then repeat step 2 until the input message sequence is retrieved.

4 The final output number is the encoded data.

6.2 Run Length Encoding

RLC also known as Run Length Coding whhich is a statistical Coding and belongs to lossless compression Coding. It is an important Coding method of raster data compression for binaries. The basic principle of run length encoding is: with a symbolic value or a string of long instead of having the same

value of consecutive symbols (continuous symbols constitute a continuous trip run length encoding hence the name), make the symbol length less than the length of the original data in all or only the columns of data code changes, a record the code and the number of the same code duplication, so as to realize the compression of data.

6.3 Bit Plane Coding

Bit plane coding is an effective technique to reduce the redundancy between pixels by processing the bit plane of image separately. It decomposes a multistage image into a series of binary images and compresses each binary image using several known binary image compression methods. Plane coding is divided into two steps: bit plane decomposition and bit plane coding. The gray level of a gray image of m bits can be expressed as a polynomial with a basis of 2.

$$a_{m-1}2^{m-1} + a_{m-2}2^{m-2} + ... + a_1 2^1 + a_0 2^0$$

According to the above formula, m coefficients of the polynomial are separated into m 1-bit bit plane, which realizes the representation of a set of multi-level gray image as a set composed of m binary images. The level 0 bit plane is the 0th bit of the original image's gray level, which is generated by the above formula. The level m-1 bit of the original image's gray level is generated by the coefficient in the above formula. Each bit plane is numbered from 0 to m-1 according to its coefficient. The value of each pixel in the bit plane is equal to the polynomial coefficient of the corresponding gray level at the corresponding position of each pixel in the original image.

7 REFERENCE

[1] "Information Theory and Coding", by Prof. K. Giridhar, pooja publications.

[2] David J. C. MacKay. *Information Theory, Inference, and Learning Algorithms* Cambridge: Cambridge University Press, 2003. ISBN 0-521-64298-1

[3] Cover, Thomas M. (2006). "Chapter 5: Data Compression".*Elements of Information Theory*. John Wiley & Sons. ISBN 0-471-24195-4.

[4] https://en.wikipedia.org/wiki/Huffman_coding

[5] www.eecs.umich.edu/courses/eecs555/lecto8.pdf

8 APPENDIX

8.1 Python code for Hoffman Coding

```
filename="A Game of Thrones.txt"
f = open(filename,'r',encoding='utf-8')
li = []
line = f.readline()
while line:
   li.append(line)
   line = f.readline()
f.close

whole_page = ""
for i in range(len(li)):
   whole_page = whole_page+li[i]

from collections import Counter
dictionary = Counter(whole_page)
dictionary = dict(dictionary)

del dictionary['\n']
del dictionary['\x0c']
del dictionary['!']
del dictionary["'"]
del dictionary["("]
del dictionary[")"]
del dictionary[","]
del dictionary["-"]
del dictionary['.']
del dictionary['0']
del dictionary['1']
del dictionary["2"];del dictionary["3"];del dictionary["4"];del
    dictionary["5"]
del dictionary["6"];del dictionary["7"];del dictionary["8"];del
    dictionary["9"]
del dictionary[":"];del dictionary["?"];del dictionary[";"];del
    dictionary["["];del dictionary["]"]


def dict2list(dic):
   keys=dic.keys()
   vals=dic.values()
   L=[(key,val) for key,val in zip(keys,vals)]
   return L
```

```
chars_freqs = dict2list(dictionary)

#Huffman Encoding

#Tree-Node Type
class Node:
   def __init__(self,freq):
      self.left = None
      self.right = None
      self.father = None
      self.freq = freq
   def isLeft(self):
      return self.father.left == self

def createNodes(freqs):
   return [Node(freq) for freq in freqs]


def createHuffmanTree(nodes):
   queue = nodes[:]
   while len(queue) > 1:
      queue.sort(key=lambda item:item.freq)
      node_left = queue.pop(0)
      node_right = queue.pop(0)
      node_father = Node(node_left.freq + node_right.freq)
      node_father.left = node_left
      node_father.right = node_right
      node_left.father = node_father
      node_right.father = node_father
      queue.append(node_father)
   queue[0].father = None
   return queue[0]



def decoder(mycode,labs,Codes):
   decode=''
   code=['' for x in range(52)]
   maxc=0;
   for i in range (len(Codes)):
      for j in range (len(Codes[i])):
         code[i]+=str(Codes[i][j])
      if maxc<=len(code[i]):
         maxc=len(code[i])
   ini=0
   for i in range (len(mycode)):
      #print(ini,i,maxc)
```

```
        if ini<=i:
            for j in range (len(code)):
                if i-ini>maxc:
                        print('error')
                if code[j]==mycode[ini:i]:
                    #print(labs[j],':', mycode[ini:i],':',code[j],':',ini,i)
                    decode+=labs[j]
                    ini=i
    return decode



def huffmanEncoding(nodes,root):
    codes = [''] * len(nodes)
    for i in range(len(nodes)):
        node_tmp = nodes[i]
        while node_tmp != root:
            if node_tmp.isLeft():
                codes[i] = '0' + codes[i]
            else:
                codes[i] = '1' + codes[i]
            node_tmp = node_tmp.father
    return codes

if __name__ == '__main__':
    chars_freqs = chars_freqs
    nodes = createNodes([item[1] for item in chars_freqs])
    root = createHuffmanTree(nodes)
    codes = huffmanEncoding(nodes,root)
    for item in zip(chars_freqs,codes):
        print('Character:%s freq:%-2d encoding: %s' %
            (item[0][0],item[0][1],item[1]))

    coding_dict = {}
    for i in range(len(codes)):
        coding_dict[chars_freqs[i][0]] = codes[i]
    coding_dict["\n"] = '111'
    del coding_dict["\n"]
    Lab = [i for i in coding_dict.keys()]
    code = []
    for i in Lab:
        st = coding_dict[i]
        code.append([i for i in st])
    enco = ''

    for i in range(len(whole_page[:1000])):
        try:
```

```
            enco = enco+coding_dict[whole_page[i]]
        except Exception:
            pass
    print('ready to encoding...')
    print('encoding: ',enco)
    print('ready to decoding...')
    print(decoder(enco,Lab,code))
```

8.2 Python code for Shannon Coding

```
import numpy as np
import math
D=2

filename="A Game of Thrones.txt"
f = open(filename,'r',encoding='utf-8')
li = []
line = f.readline()
while line:
    li.append(line)
    line = f.readline()
f.close

whole_page = ""
for i in range(len(li)):
    whole_page = whole_page+li[i]

from collections import Counter
dictionary = Counter(whole_page)
dictionary = dict(dictionary)

del dictionary['\n']
del dictionary['\x0c']
del dictionary['!']
del dictionary["'"]
del dictionary["("]
del dictionary[")"]
del dictionary[","]
del dictionary["-"]
del dictionary['.']
del dictionary['0']
del dictionary['1']
del dictionary["2"];del dictionary["3"];del dictionary["4"];del
    dictionary["5"]
```

```
del dictionary["6"];del dictionary["7"];del dictionary["8"];del
    dictionary["9"]
del dictionary[":"];del dictionary["?"];del dictionary[";"];del
    dictionary["["];del dictionary["]"]



def dict2list(dic):
   keys=dic.keys()
   vals=dic.values()
   L=[(key,val) for key,val in zip(keys,vals)]
   return L
chars_freqs = dict2list(dictionary)

freq=[]
Data=[]
lab=[]
Lab=[]
All=0;
for i in range(52):
   All+=chars_freqs[i][1]
for i in range(52):
   lab+=[chars_freqs[i][0]]
   freq+=[chars_freqs[i][1]/All]
data=list(zip(lab,freq))
index=np.argsort(freq, 0)[:]
for i in range(52):
   Data+=[data[index[51-i]][1]]
   Lab+=[lab[index[51-i]][0]]


def Add(l):
   add=0
   for i in range (len(l)):
      add+=l[i]
   return add
def dec2bin(x):
   if x==0:
      return [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0]
   else:
      x-= int(x)
      bins = []
      while x:
         x *= 2
         bins.append(1 if x>=1. else 0)
```

```
            x -= int(x)
        return bins

cumulative=[]
length=[]
code=[]
for i in range(52):
    cumulative+=[Add(Data[0:i])]
    length+=[math.ceil(-math.log(Data[i],D))]

    code+=[dec2bin(cumulative[i])[0:length[i]]]
    print(Lab[i],'s code is:',code[i],":",length[i],cumulative[i])
    print('----------------------')

def encodeing(filename,Lab,code):
    f = open(filename,'r',encoding='utf-8')
    li = []
    line = f.readline()
    while line:
        li.append(line)
        line = f.readline()
    f.close
    Final_code=''
    for j in range(len(li)):
        for i in range (len(li[j])):
            for k in range(len(Lab)):
                if Lab[k]==li[j][i]:
                    for n in range(len(code[k])):
                        Final_code+=str(code[k][n])
    return Final_code


filename="A Game of Thrones.txt"
mycode=encodeing(filename,Lab,code)


def decoder(mycode,labs,Codes):
    decode=''
    code=['' for x in range(52)]
    maxc=0;
    for i in range (len(Codes)):
        for j in range (len(Codes[i])):
            code[i]+=str(Codes[i][j])
        if maxc<=len(code[i]):
            maxc=len(code[i])
    ini=0
    for i in range (len(mycode)):
```

```
      #print(ini,i,maxc)
      if ini<=i:
         for j in range (len(code)):
            if i-ini>maxc:
                  print('error')
            if code[j]==mycode[ini:i]:
               #print(labs[j],':', mycode[ini:i],':',code[j],':',ini,i)
               decode+=labs[j]
               ini=i
   return decode

out=decoder(mycode,Lab,code)
print(mycode[0:1000])
print(out[0:1000])
```

8.3 Python code for Fano Coding

```
import numpy as np
import math

filename="A Game of Thrones.txt"
f = open(filename,'r',encoding='utf-8')
li = []
line = f.readline()
while line:
   li.append(line)
   line = f.readline()
f.close

whole_page = ""
for i in range(len(li)):
   whole_page = whole_page+li[i]

from collections import Counter
dictionary = Counter(whole_page)
dictionary = dict(dictionary)

del dictionary['\n']
del dictionary['\x0c']
del dictionary['!']
del dictionary["'"]
del dictionary["("]
del dictionary[")"]
del dictionary[","]
del dictionary["-"]
```

```
del dictionary['.']
del dictionary['0']
del dictionary['1']
del dictionary["2"];del dictionary["3"];del dictionary["4"];del
    dictionary["5"]
del dictionary["6"];del dictionary["7"];del dictionary["8"];del
    dictionary["9"]
del dictionary[":"];del dictionary["?"];del dictionary[";"];del
    dictionary["["];del dictionary["]"]


def dict2list(dic):
   keys=dic.keys()
   vals=dic.values()
   L=[(key,val) for key,val in zip(keys,vals)]
   return L
chars_freqs = dict2list(dictionary)

freq=[]
Data=[]
lab=[]
Lab=[]
All=0;
for i in range(52):
   All+=chars_freqs[i][1]
for i in range(52):
   lab+=[chars_freqs[i][0]]
   freq+=[chars_freqs[i][1]/All]
data=list(zip(lab,freq))
index=np.argsort(freq, 0)[:]
for i in range(52):
   Data+=[data[index[i]][1]]
   Lab+=[lab[index[i]][0]]

def Add(l):
   add=0
   for i in range (len(l)):
      add+=l[i]
   return add
def dec2bin(x):
   if x==0:
      return [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
   else:
      x-= int(x)
```

```
      bins = []
      while x:
         x *= 2
         bins.append(1 if x>=1. else 0)
         x -= int(x)
      return bins

def get_cumulative(Data):
   cumulative=[]
   for i in range(len(Data)):
      cumulative+=[Add(Data[0:i+1])]
   return cumulative

cumulative=get_cumulative(Data)
print(cumulative )

def find_middle(l,da):
   av=sum(da)/2
   for i in range(len(l)):
      l[i]=abs(l[i]-av)
   return l.index(min(l))
find_middle(cumulative,Data)

def finding_cut(Data,cut,all_data):
   n=100
   for s in range(len(all_data)):
      if Data[0]==all_data[s]:
         n=s
   if n==100:
      print('error')

   D=get_cumulative(Data)
   if n==0:
      cut+=[find_middle(D,Data)]
   else:
      #print('initial from: ',n,'part
          cut=',find_middle(D,Data),'add:',find_middle(D,Data)+n+1)
      cut+=[find_middle(D,Data)+n+1]
   #print(len(cut))
   if len(cut)==1:
      cut=finding_cut(all_data[0:cut[0]],cut,all_data)
      if len(cut)==51:
               #print('return')
               return cut
      #print('only happen once')
   elif len(cut)<51:
      for j in range(0,len(cut)-1):
```

```
            check=sorted(cut)
            if check[0]-0>1:
               #print('cut:',cut)
               #print('j:',j+1,j)
               #print( 'check:',check[j+1],check[j])
               #print('con1 reinput:',0,check[0])
               cut=finding_cut(all_data[0:check[0]],cut,all_data)
               if len(cut)==51:
                  #print('return')
                  return cut
            elif 52-check[len(check)-1]>1:
               #print('cut:',cut)
               #print('j:',j+1,j)
               #print( 'check:',check[j+1],check[j])
               #print('con2 reinput:',check[len(check)-1],51)
               cut=finding_cut(all_data[check[len(check)-1]:51],cut,all_data)
               if len(cut)==51:
                  #print('return')
                  return cut
            if check[j+1]-check[j]>1:
               #print('cut:',cut)
               #print('j:',j+1,j)
               #print( 'check:',check[j+1],check[j])
               #print('con3 reinput:',check[j],check[j+1])
               cut=finding_cut(all_data[check[j]:check[j+1]],cut,all_data)
               if len(cut)==51:
                  #print('return')
                  return cut
      elif len(cut)==51:
         print('return')
         return cut

cut=[]
cuts=finding_cut(Data,cut,Data)

def find_the_cloest(i,cut):
      small=0;
      big=52;
      if i==0:
         return small, big
      else :
         for j in range(i):
            if cut[j]<cut[i] and cut[j]>=small:
               small=cut[j]
            if cut[j]>cut[i] and cut[j]<=big:
               big=cut[j]
      return small, big
```

```
def code_generator(code,cut):
   for i in range (len(cut)):
      s,b = find_the_cloest(i,cut)
      for j in range(s,cut[i]):
         code[j]+=[0]
      for k in range(cut[i],b):
         code[k]+=[1]
   return code

code=[[] for x in range(52)]
codes=code_generator(code,cuts)

for i in range(52):
   print(Lab[i],'s code is:',codes[i])
   print('----------------------')


def shannon_codeing(filename,Lab,code):
   f = open(filename,'r',encoding='utf-8')
   li = []
   line = f.readline()
   while line:
      li.append(line)
      line = f.readline()
   f.close
   Final_code=''
   for j in range(len(li)):
      for i in range (len(li[j])):
         for k in range(len(Lab)):
            if Lab[k]==li[j][i]:
               for n in range(len(code[k])):
                  Final_code+=str(code[k][n])
   return Final_code

filename="A Game of Thrones.txt"
mycode=shannon_codeing(filename,Lab,codes)
```