**Assignment 3**

**Group:**BDA-1904

**Student Name:**Berik Gulina &Abdikalyk Gulnazym

**Github Link:** <https://github.com/gulina0426/InformationTheory>

**1-Description:**

Firstly,during the whole work we have totally 5 tasks are going to implemented.And the whole task we are going to working on the python code with the algorithm what we have chose is “Huffman” algorithm.As we know that the first task is like the continue part with the assignment 2,then the 2 to 4 is the decoding part,the task 5 is just going to calculate the ratio.

**2-Team working process:**

As usual we have done the whole work in the colab and teams to see each others working process and solving each problems together.We have divided equally the 1,3 was done by Gulnazym and 2,4 was done by Gulina then the last 5 one we have finished together.Then in the working process if we have any problems or other things we also helped to solve together and trying to understand.

**3-Execution part and screenshort:**

**3.1 Task-1:**

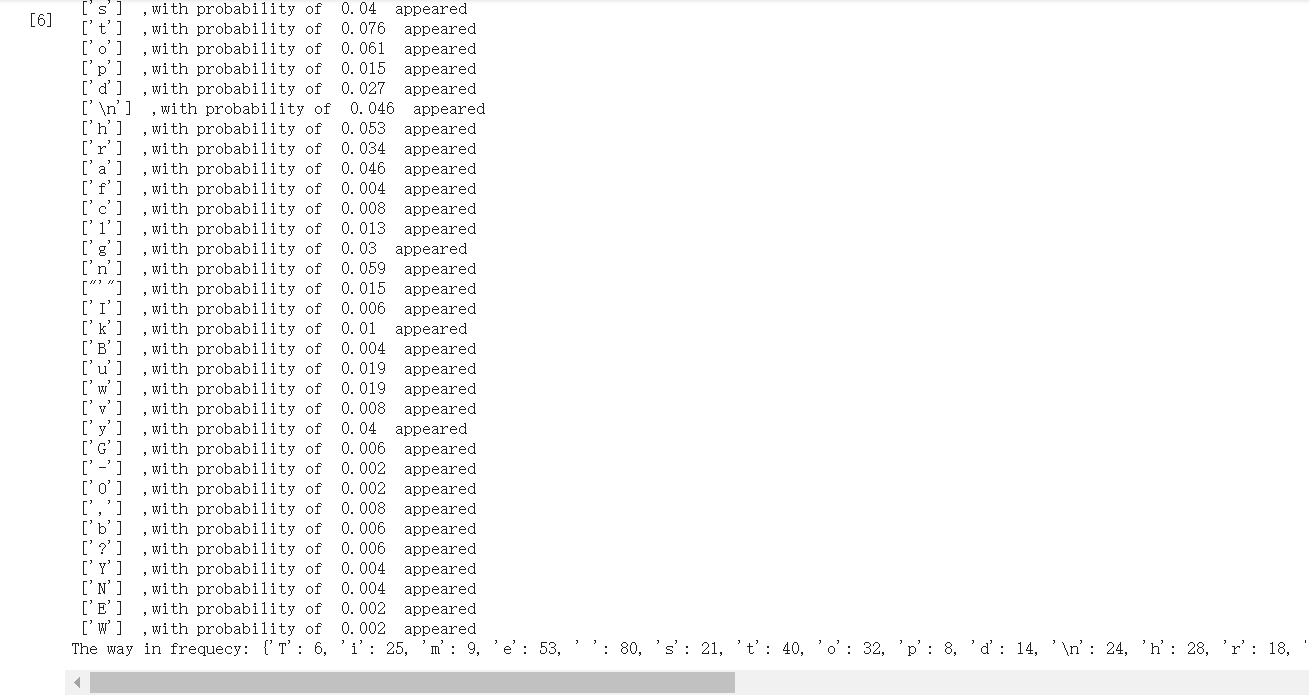
****

Figure-1 Result of the first task

**Source code-1:**

from collections import Counter

#Firstly set our file in order to be readable

FileeRead = open('abc.txt','r',encoding='utf-8').read()

#Then after that we set the total length

TotalCount=len(FileeRead)

#Our first function here inorder to count the whole

def GulCalculation\_1():

G\_Count\_Found ={}#Initial

for i in FileeRead:#Started the loop statement

if i in G\_Count\_Found:

G\_Count\_Found[i]+=1#If we found the matched then will be added into the counter

else:

G\_Count\_Found[i]=1#If not then just remain

for i in G\_Count\_Found:#Then are going to show the related message output and roundd to 3 var

print(f' {[i[0]]}  ,with probability of  {round(G\_Count\_Found[i] / TotalCount,3)}  appeared  ')

#Here is the second way for present the frequency by using the set default

def GulCalculation\_2():

G\_Count\_Found\_2 ={}

for char in FileeRead:

G\_Count\_Found\_2.setdefault(char,0)

# print(count.setdefault(char,0))

G\_Count\_Found\_2[char]+=1

print("The way in frequecy:",G\_Count\_Found\_2)

if \_\_name\_\_ =='\_\_main\_\_':

GulCalculation\_1()

GulCalculation\_2()

**3.2 Task-2:**

****

Figure-2 Result of the second task

**Source code-2:**

class Node(object):#So firstly we have the class which is about the node

def \_\_init\_\_(self, data):

self.data = data#It has the parameter with the data

self.g\_parent = None#It has the parameter with the parent

self.g\_left\_child = None#It has the parameter with the left children

self.g\_right\_child = None#It has the parameter with the right children

self.g\_is\_in\_tree = False#Then to check if it is existed in our tree

class HuffmanTree(object):#Then to build our huffman tree class

def \_\_init\_\_(self):

self.\_\_root = None#We are going to present the tree so each parameter defines the specific vls

self.g\_prefix\_branch = '├'

self.g\_prefix\_trunk = '|'

self.g\_prefix\_leaf = '└'

self.g\_prefix\_empty = ''

self.g\_prefix\_left = '─L─'

self.g\_prefix\_right = '─R─'

def is\_empty(self):#This one is going to check if empty or noe

return not self.\_\_root

@property

def root(self):

return self.\_\_root

    @root.setter

    def root(self, value):#This process is going to check our not

    self.\_\_root = value if isinstance(value, Node) else Node(value)

    def show\_tree(self):#Here is going to show our tree if it is empty or not

    if self.is\_empty():#If empty then just print empty

    print('Just\_Empty')

    return#Else is going to print

    print('-' \* 20)

        print(self.\_\_root.data)

        self.\_\_print\_tree(self.\_\_root)

        print('-' \* 20)

    def \_\_print\_tree(self, node, prefix=None):#Here is our function for print the tree

        if prefix is None:#If it is none then nothing

            prefix, prefix\_left\_child = '', ''

        else:#If it is not null then going to replace and print each

            prefix = prefix.replace(self.g\_prefix\_branch, self.g\_prefix\_trunk)

            prefix = prefix.replace(self.g\_prefix\_leaf, self.g\_prefix\_empty)

            prefix\_left\_child = prefix.replace(self.g\_prefix\_leaf, self.g\_prefix\_empty)

        if self.has\_child(node):#If it has the child then going to consider 2 conditions if none or not

            if node.g\_right\_child is not None:#If the right child which contains the node is not null

                print(prefix + self.g\_prefix\_branch + self.g\_prefix\_right + str(node.g\_right\_child.data))

                if self.has\_child(node.g\_right\_child):#If it also contains the each child

                    self.\_\_print\_tree(node.g\_right\_child, prefix + self.g\_prefix\_branch + ' ')

            else:#Else going to print the other condt.

                print(prefix + self.g\_prefix\_branch + self.g\_prefix\_right)

            if node.g\_left\_child is not None:#Same conditions consider with the left children

                print(prefix + self.g\_prefix\_leaf + self.g\_prefix\_left + str(node.g\_left\_child.data))

                if self.has\_child(node.g\_left\_child):#If it also has the chid

                    prefix\_left\_child += '  '

                    self.\_\_print\_tree(node.g\_left\_child, self.g\_prefix\_leaf + prefix\_left\_child)

            else:#else is going to print the others with itself as well

                print(prefix + self.g\_prefix\_leaf + self.g\_prefix\_left)

    def has\_child(self, node):

        return node.g\_left\_child is not None or node.g\_right\_child is not None

    def huffman(self, leavers):

       #Then wee need to build our huffman tree in here

        if len(leavers) <= 0:#if our leavers are less and equal to zero

            return#then return the value

        if len(leavers) == 1:#if is one then none and return the ndoe

            self.root = Node(leavers[0])

            return

        woods = list()#if is exisiting with the list which we provided

        for i in range(len(leavers)):

            woods.append(Node(leavers[i]))#Then going to add it

        while len(woods) < 2\*len(leavers) - 1:

            node1, node2 = Node(float('inf')), Node(float('inf'))

            for j in range(len(woods)):

                if node1.data > node2.data:

                    node1, node2 = node2, node1

                if woods[j].data < node1.data and woods[j].g\_is\_in\_tree is False:

                    node1, node2 = woods[j], node1

                elif node1.data <= woods[j].data < node2.data and woods[j].g\_is\_in\_tree is False:

                    node2 = woods[j]

            parent\_node = Node(node1.data + node2.data)

            woods.append(parent\_node)

            parent\_node.g\_left\_child, parent\_node.g\_right\_child = node1, node2

            self.root, node1.g\_parent, node2.g\_parent = parent\_node, parent\_node, parent\_node

            node1.g\_is\_in\_tree, node2.g\_is\_in\_tree = True, True

if \_\_name\_\_ == '\_\_main\_\_':

    tree = HuffmanTree()

    leavers = [10, 5, 7, 13, 17, 11]#The list we provided for example

    tree.huffman(leavers)

tree.show\_tree()

**3.3 Task-3:**

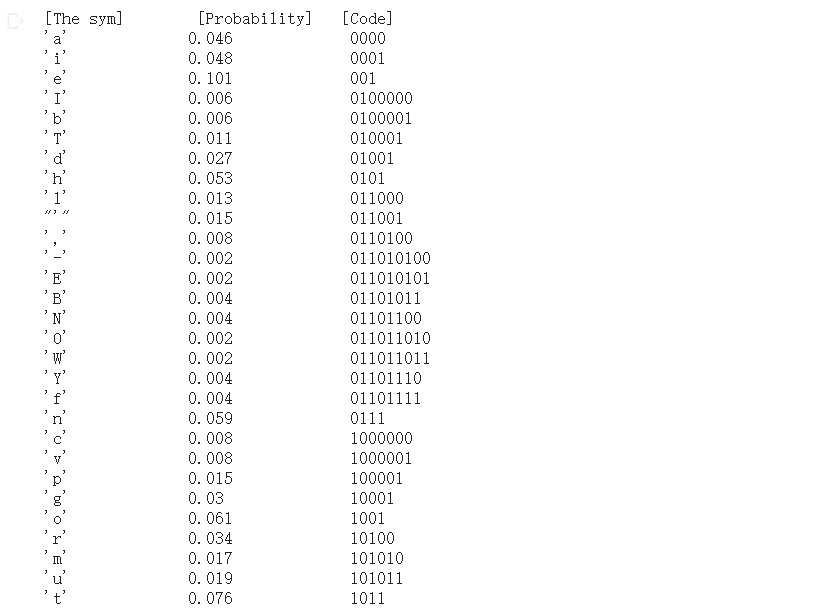
****

Figure-3 Result of the third task

**Source code-3:**

import heapq

#Firstly we have define our function to calculate the frequency

def G\_The\_frequency(file1):

g\_saving={}#set the saving part first

for line in file1:#Then to find each in the each line

for i in range(len(line)):#If we find the related line then going to add in the saving

if not(line[i] in g\_saving):

g\_saving[line[i]]=1

else:#Else will be the converse

g\_saving[line[i]]+=1

return (g\_saving)

def G\_Part\_Encode(text):#Then secondly is our encoding part

exchange=[]

for g\_symbollic,g\_freq in text.items():#In the text items we need the symbol and frequency

exchange.append([g\_freq,[g\_symbollic,'']])

heapq.heapify(exchange)#then going to exchange

while len(exchange)>1:

left=heapq.heappop(exchange)

right=heapq.heappop(exchange)

for element in left[1:]:

element[1]='0'+element[1]

for element in right[1:]:

element[1]='1'+element[1]

heapq.heappush(exchange,[left[0]+right[0]]+left[1:]+right[1:])

exchange=heapq.heappop(exchange)

return exchange

if \_\_name\_\_ == "\_\_main\_\_":#Here is our main function

Thefilename="abc.txt"#read the file

file=open(Thefilename,'r',encoding='utf-8').read()

The\_total = len(file)#total is our length out file

g\_freq=G\_The\_frequency(file)#frequency is our freq of file

encoded=G\_Part\_Encode(g\_freq)#encode is our code part of file

print("[The sym] \t [Probability] \t [Code]")#Then going to print

for i in encoded[1:]:

count=round(g\_freq[i[0]]/ The\_total,3)

print(repr(i[0])+"\t\t"+ str(count) +"\t\t  "+ i[1])

**3.4 Task-4:**

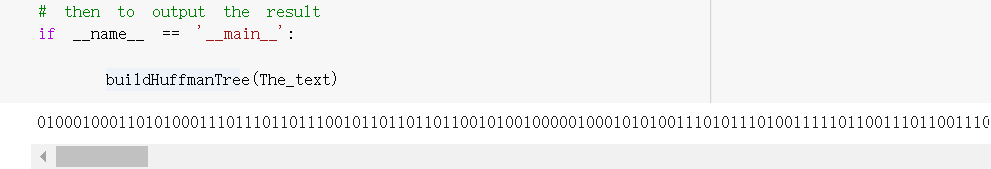
****

Figure-4 Result of the forth task

**Source code-4:**

import os

import heapq

from heapq import heappop, heappush

#Firstly set our file in order to be readable

The\_text = open('abc.txt','r',encoding='utf-8').read()

def Check\_isLeaf\_g(root):#Set our function to check is the leaf or not

    return root.gleft is None and root.gright is None#then to return none or not

    # here start the tree node

class Node:#Then is our class about the Node

    def \_\_init\_\_(self, gch, gfreq, gleft=None, gright=None):

        self.gch = gch#Here we need the para with children,frequency,left and right

        self.gfreq = gfreq#set each of them related to their

        self.gleft = gleft

        self.gright = gright

    # Here we need to override the whole function to make our node class work with priority in queue

    # such in here that the highest priority item who has the lowest frequency

    def \_\_lt\_\_(self, other):#Then in here we need to define this one

        return self.gfreq < other.gfreq #Then to return the part

# Here need to traverse the our Tree and store it in the dictionary

def encode(root, str, huffman\_code):

    if root is None:#If none then return

        return

    # In here need to found a leaf node as well

    if Check\_isLeaf\_g(root):#Check the related condtion as follow

        huffman\_code[root.gch] = str if len(str) > 0 else '1'

    encode(root.gleft, str + '0', huffman\_code)#our left child

    encode(root.gright, str + '1', huffman\_code)#our right child

# decoding part with our tree

def decode(root, index, str): #define it

        if root is None:#if none then return index

            return index

        # Here need to found a leaf node

        if Check\_isLeaf\_g(root):

            print( end='')

            return index

        index = index + 1#then the index will be added

        root = root.gleft if str[index] == '0' else root.gright

        return decode(root, index, str) #then return the decoding part

    # here need to build the tree

def buildHuffmanTree(The\_text):#The text is our file which will uploaded

    # if the empty string with condition

    if len(The\_text) == 0:

        return

    # here we need to firstly count the freq with each char,then to store it

    gfreq = {i: The\_text.count(i) for i in set(The\_text)}#counting the freq

    # here need to store the live nodes with creating the queue

    pq = [Node(k, v) for k, v in gfreq.items()]

    heapq.heapify(pq)

    while len(pq) != 1:

        # Remove the two nodes of the highest priority

        gleft = heappop(pq)

        gright = heappop(pq)

      #here we need to create the new one

        total = gleft.gfreq + gright.gfreq

        heappush(pq, Node(None, total, gleft, gright))

    # the root in here will stores the pointter of the tree

    root = pq[0]

    # traverse our\_tree and to store it in the dicc

    huffmanCode = {}

    encode(root, "", huffmanCode)

    # print the resultt

    str = ""

    for c in The\_text:

        str += huffmanCode.get(c)

    print(str)#saving it to the file

    text\_file\_save = open("outputtxt.txt", "w")

    nnn = text\_file\_save.write(str)

    text\_file\_save.close()

    if Check\_isLeaf\_g(root):

        # here is the special carses

        while root.gfreq > 0:

            print( end='')

            root.gfreq = root.gfreq - 1

    else:

        #decode and encode the string part here

        index = -1

        while index < len(str) - 1:

            index = decode(root, index, str)

# then to output the result

if \_\_name\_\_ == '\_\_main\_\_':

buildHuffmanTree(The\_text)

**3.5 Task-5:**

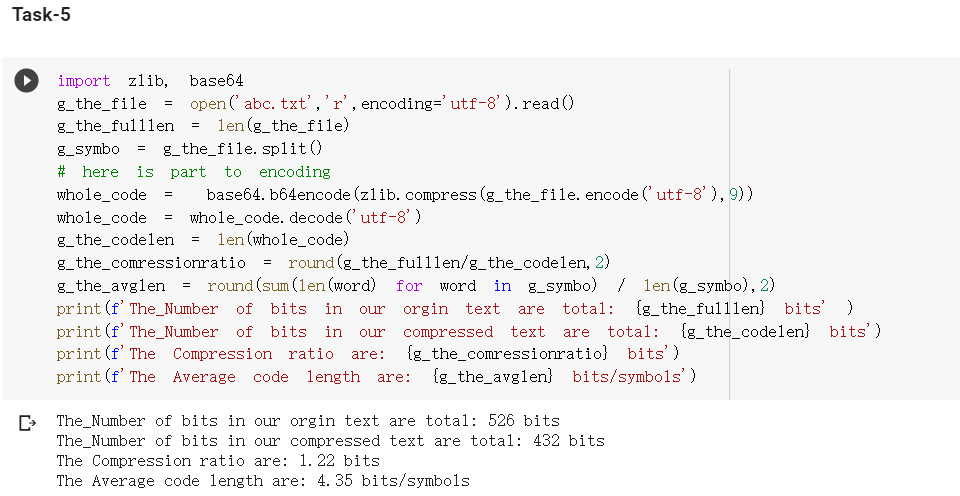
****

Figure-5 Result of the fifth task

**Source code-5:**

import zlib, base64

g\_the\_file = open('abc.txt','r',encoding='utf-8').read()

g\_the\_fulllen = len(g\_the\_file)

g\_symbo = g\_the\_file.split()

# here is part to encoding

whole\_code =  base64.b64encode(zlib.compress(g\_the\_file.encode('utf-8'),9))

whole\_code = whole\_code.decode('utf-8')

g\_the\_codelen = len(whole\_code)

g\_the\_comressionratio = round(g\_the\_fulllen/g\_the\_codelen,2)

g\_the\_avglen = round(sum(len(word) for word in g\_symbo) / len(g\_symbo),2)

print(f'The\_Number of bits in our orgin text are total: {g\_the\_fulllen} bits' )

print(f'The\_Number of bits in our compressed text are total: {g\_the\_codelen} bits')

print(f'The Compression ratio are: {g\_the\_comressionratio} bits')

print(f'The Average code length are: {g\_the\_avglen} bits/symbols')