Mini-Project. Performance and Reliability Analysis of Communication  
Networks

Task 1. Moby Dick is an extremely long novel, and the size of the plain-text file containing the unabridged version is approximately 1.2 MB.  
(see https://gutenberg.org/ebooks/2701). However, language is relatively easy to compress, as it is very redundant.

* Encode the novel with Huffman and Shannon-Fano compression, using individual  
  characters as symbols and estimating their probability by their relative frequency.  
  You can use any programming language to implement both the codes and the  
  probability estimation.
* Encode the novel using words as symbols (you can count punctuation and white  
  spaces as separate words), then run the encoding again - how has the size of the  
  file changed?
* Implement a version of LZW or LZ77 and encode the book.

Your report should provide both the code listing for the compression algorithms and a  
short discussion of the results you obtained

# Shannon and Huffman -Algorithm core functionalities

## The Entropy calculation for both algorithms

The entropy is calculated using This code module is illustrated and explained below

def Entropy(*Novel*):

    UniqueCharacters = set(*Novel*) 🡨 first we find the all the unique characters in this Novel

    NumberOfUniqueCharacters = len(UniqueCharacters) 🡨 Taking the length we how many there are

    MobyDockNrOfCarakters = len(*Novel*) 🡨 we find out how many characters there are in the novel

    Hx = 0 🡨 Hx will hold the value for H(x)

    EntropyList = [] 🡨 will hold prob of character and probability of character

    for carakter in UniqueCharacters: 🡨 walks through all the unique characters

        NumberOfSertantCharInNovel = *Novel*.count(carakter) 🡨 the amount of in the Novel

        ProbOfCarakter = NumberOfSertantCharInNovel/MobyDockNrOfCarakters 🡨 pr[X=x]

        Pruduct = ProbOfCarakter \* numpy.log(1/ProbOfCarakter) 🡨

        Hx += Pruduct 🡨

        EntropyList.append([carakter,ProbOfCarakter])

    EntropyList = sorted(EntropyList,*key*=lambda *x*: *x*[1])

    return Hx, EntropyList

## Shannon core funktionalitet

The idea is to take the novel and split it into branches , and find the branches that is only ‘1’ Character and then find the code for this element

def SHANNON(*EntropyList*, *prefix*=""):

    if len(*EntropyList*) == 1: 🡨 when we encounter a branch that is only ‘1’ (character) big we return it

        symbol = *EntropyList*[0][0]

        return {symbol: *prefix*}

All below is the case when we have a branch that is not ‘1’ (character) but bigger

    split\_idx = HaffPoint(*EntropyList*) 🡨 we split the list so we get the half point closest to 50%

    left = *EntropyList*[:split\_idx + 1] 🡨 we denote the one side left

    right = *EntropyList*[split\_idx + 1:] 🡨 and the other one right

    codes = {}

    codes.update(SHANNON(left, *prefix* + "0"))  *🡨 we update code with a “1” and call this function for the left part*

    codes.update(SHANNON(right, *prefix* + "1"))  *🡨 same for the right*

    return codes

## Huffman core functionalities

def EvalueateHuffmand(*EntropyList*, *node*, *val*=''):

    total = 0 🡨 This will hold the value for the Entropy

    newVal = *val* + str(*node*.huff) 🡨 Updates the three that gets smaller and smaller

    if(*node*.left):

        total += EvalueateHuffmand(*EntropyList*, *node*.left, newVal) 🡨 we adds 0 to the branches below

    if(*node*.right):

        total += EvalueateHuffmand(*EntropyList*, *node*.right, newVal) 🡨 we adds 1 to the branches below

    if(not *node*.left and not *node*.right): 🡨 when final list

*#print(f"{node.symbol} -> {newVal}")*

        for i in *EntropyList*:

            if i[0] == *node*.symbol: 🡨 calculate the value for all Characters

*val* = i[1] \* len(newVal)

                total += *val*

    return total 🡨 The E[L] value

## Results for Huffman and Shannon

The results of the encoding can be seen below

Table Shannon Characters selected this is the terminal output

|  |
| --- |
| Aalborg Universitet/GIT/Mini-Project NetPerf 2024/Shannon Encoder/Shannon.py"  Shannon  Character Chosen  Working On Entropy Calculations ...  Entropy: 3.1169  Working on SHANNON Algorithm ...  Shannon Codes: can be shown in [def EvalueateShannon]  Algorithm Entropy : 4.538180056128736 Entropy: 3.1169 |

Table Shannon Words selected this is the terminal output

|  |
| --- |
| Aalborg Universitet/GIT/Mini-Project NetPerf 2024/Shannon Encoder/Shannon.py"  Shannon  Words Chosen  Working On Entropy Calculations ...  Entropy: 4.8073  Working on SHANNON Algorithm ...  Shannon Codes: can be shown in [def EvalueateShannon]  Algorithm Entropy : 7.065208721827754 Entropy: 4.8073 |

Table huffman Characters selected this is the terminal output

|  |
| --- |
| Aalborg Universitet/GIT/Mini-Project NetPerf 2024/Huffmand Encoder/Huffmand.py"  Huffman  Character Chosen  Working On Entropy Calculations ...  Entropy: 3.1169  Working on HUFFMAND Algorithm ...  Huffman Codes: can be shown in [def EvalueateHuffmand]  Algorithm Entropy : 4.526056249873811 Entropy: 3.1169 |

Table huffman Characters selected this is the terminal output

|  |
| --- |
| Aalborg Universitet/GIT/Mini-Project NetPerf 2024/Huffmand Encoder/Huffmand.py"  Huffman  Words Chosen  Working On Entropy Calculations ...  Entropy: 4.8073  Working on HUFFMAND Algorithm ...  Huffman Codes: can be shown in [def EvalueateHuffmand]  Algorithm Entropy : 6.999930934764544 Entropy: 4.8073 |

We see what we as expected that Huffman is a little better than Shannon in general. And that character is better than words because there in a normal use case like the is a lot fewer characters than words.

## LZW

def LZW\_Homebrew(*Alphabet*, *Data*):

    list = *Alphabet*

    Novel = *Data*

    i = 0 🡨 calculate we stand at

    j = 0 🡨 holds old values of i

    code = [] 🡨 Where the code will be sored

    while 1: 🡨 while we not done

        j = i 🡨 we update j to match j

        while 1: 🡨 while we haven’t fount a match in the alphabet

            lzw = ''.join(Novel[j:i+1]) 🡨 the character string we trying to macth

            if lzw not in list: 🡨 is it not in alpabet ad it

                list.append(lzw)

                code.append(list.index(lzw[:-1])) 🡨 we append the old code ((names) append (name))

                j = i 🡨 updates the steps we have taken

                break

            i += 1 🡨 if it was or wasn’t in alphabet we move one to the next character

            if lzw == ''.join(Novel[j:]): 🡨 if this is the end alpabet append to code

                code.append(list.index(lzw))

                break

*# print("----------")*

*# print(list)*

*# print(i)*

*# print("----------")*

        print(round((i/(len(Novel)-1))\*100, 3), "\t %\r", *end*="") 🡨 progress bar

        if i >= len(Novel)-1: 🡨 are we done with the novel break

            break

    code.append(0) 🡨 end character

*#print(code)*

    f = open("Homebrew LZW.txt","w") 🡨 write to text file.

    f.write(str(code))

    f.close()

### Results of LZW

|  |
| --- |
| The string to compress “TOBEORNOTTOBEORTOBEORNOT “  Aalborg Universitet/GIT/Mini-Project NetPerf 2024/LZW Encoder/LZW.py"  working on OPtimized LZW: ...  working on HOMEBREW LZW: ...  Homebrew LZW.txt  gives [20, 15, 2, 5, 15, 18, 14, 15, 20, 27, 29, 31, 36, 30, 32, 34, 0]  Optimized LZW.tzt  gives [21, 16, 3, 6, 16, 19, 15, 16, 21, 28, 30, 32, 37, 31, 33, 35, 0]  This is because it adds the “” none Caracter, to the alphabet this will come in handy later.  This it to prove that the models work (at least for this string) |

We believe the algorithm works but that The novel is not big enough or that there is to many special cases where it doesn’t repeat

The novel is 1,247 KB

The LZW encoded is 1,609 KB