Data_Compression_Project_LZ_1

November 14, 2022

0.1 ECE 231A : Data Compression Project Module 2

Please follow our instructions in the same order and print out the entire results and codes when completed.

[96]:
```
import numpy as np
import matplotlib.pyplot as plt
import csv
from collections import Counter
```

[97]:
```
##############################################################################################
# Read Text file
##############################################################################################
f = open("Toy_Example_LZ.txt", "r")
text = f.read()
print(text)
```

```
AAAABBBBBBAAAAACCCCCEEEEAAAADDDDBBBAAACCCDDDEEEEEBBBCCCC
```

[98]:
```
##############################################################################################
# Compute the distribution
##############################################################################################
def compute_distribution(text):
    """
    Inputs:
    - text: A string containing the text to be encoded.

    Returns:
    - symbols: a list of tuples of the form (char,prob), where char is a␣
 ↪character appears in the text
                and prob is the number of times this character appeared in text␣
 ↪divided by the length of text.
    - entropy: a number represnting the entropy of the source symbols
    """
    # ============================================================ #
    # YOUR CODE HERE:
    # ============================================================ #
    counter = dict()
    for k in text:
```

```
        if k in counter:
            counter[k]+=1/len(text)
        else:
            counter[k]=1/len(text)

    symbols = []
    entropy = 0
    for j in counter.keys():
        symbols.append((j, counter[j]))
        entropy-=np.log2(counter[j])*counter[j]
    symbols = sorted(symbols, key=lambda x:x[1])

    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
    return symbols, entropy

symbols, entropy = compute_distribution(text)
print(symbols)
print(entropy)
```

```
[('D', 0.12499999999999997), ('E', 0.16071428571428567), ('B',
0.21428571428571422), ('C', 0.21428571428571422), ('A', 0.28571428571428564)]
2.267713681259536
```

0.2 Binary LEMPEL-ZIV Coding

[99]:
```
##########################################################################################
# Initialize the dictionary of both the sender and the receiver
##########################################################################################
def intialize_dict(symbols):
    """
    Inputs:
    - symbols: a list of tuples of the form (char,prob), where char is a␣
 ↪character appears in the text
               and prob is the number of times this character appeared in text␣
 ↪divided by the length of text.
    Returns:
    - TX_dictionary: dictionary containing the symbols in symbols and its␣
 ↪corresponding binary code
    - RX_dictionary: dictionary containing the symbols in symbols and its␣
 ↪corresponding binary code
    """
    TX_dictionary = dict()
    RX_dictionary = dict()
    # ============================================================ #
    # YOUR CODE HERE:
```

```
    # ============================================================ #
    counter = 0
    l = 0
    c = len(symbols)
    while (c):
        c=c//2
        l+=1
    for i in symbols:
        TX_dictionary[i[0]]= format(counter, '0'+str(l)+'b')
        RX_dictionary[i[0]]= format(counter, '0'+str(l)+'b')
        counter+=1
    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
    return TX_dictionary,RX_dictionary

TX_dictionary, RX_dictionary = intialize_dict(symbols)
print(TX_dictionary, RX_dictionary)
```

```
{'D': '000', 'E': '001', 'B': '010', 'C': '011', 'A': '100'} {'D': '000', 'E':
'001', 'B': '010', 'C': '011', 'A': '100'}
```

[100]:
```
##########################################################################################
#Encode the text
##########################################################################################
def Lempel_ziv_coding(text, TX_dictionary):
    """
    Inputs:
    - text: A string containing the text to be encoded.
    - TX_dictionary: Initialized decitionary of the sender
    Returns:
    - TX_dictionary: the updated dictionary of the sender.
    - Code: the code of the input text
    """
    code =  ''
    # ============================================================ #
    # YOUR CODE HERE:
    # ============================================================ #
    while (len(text)>0):
        subseq = text[0]
        counter = 1
        while subseq in TX_dictionary:
            subseq += text[counter]
            counter+= 1
            if (counter == len(text)):
                break
        if (not subseq in TX_dictionary):
```

```
            c = TX_dictionary[subseq[:-1]]
            l = len(TX_dictionary)
            nextCode = bin(l)[2:]
            code += c+TX_dictionary[subseq [-1]]
            if (l & (l-1) == 0):
                for i in TX_dictionary.keys():
                    TX_dictionary[i] = "0"+TX_dictionary[i]
            TX_dictionary[subseq] = nextCode
            text = text[len(subseq):]
        else:
            code += TX_dictionary[subseq]
            return



    # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #
    return TX_dictionary, code
TX_dictionary, code = Lempel_ziv_coding(text, TX_dictionary)
print(TX_dictionary)
print("The code of the text is",code)
```

```
{'D': '00000', 'E': '00001', 'B': '00010', 'C': '00011', 'A': '00100', 'AA':
'00101', 'AAB': '00110', 'BB': '00111', 'BBB': '01000', 'AAA': '01001', 'AAC':
'01010', 'CC': '01011', 'CCE': '01100', 'EE': '01101', 'EA': '01110', 'AAAD':
'01111', 'DD': '10000', 'DB': '10001', 'BBA': '10010', 'AACC': '10011', 'CD':
'10100', 'DDE': '10101', 'EEE': '10110', 'EB': '10111', 'BBC': '11000', 'CCC':
'11001'}
The code of the text is 1001001010100100101110100101010001010011001100111011000100010001000101001001000000000000000000000100011100100010100001100011000001000000001011010000100001000100011100011010110001100
```

[101]:
```
##########################################################################################
# Compute the expected length of the Lempel-Ziv code
##########################################################################################
def compute_expected_length(text, code):
    """
    Inputs:
    - text: input text.
    - code: the code of the input text
    Returns:
    - Expected_length: a number represents the expected length of Lempel-Ziv␣
 ↪code per sample
    """
    Expected_length = 0
    # ============================================================ #
```

```
    # YOUR CODE HERE:
    # ================================================================ #
    Expected_length =  len(code)/len(text)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return Expected_length

Expected_length = compute_expected_length(text, code)
print("Expected length of the Lempel-Ziv code:   ", Expected_length)
```

```
Expected length of the Lempel-Ziv code:    3.1785714285714284
```

[102]:
```
##########################################################################################
# Decode a text
##########################################################################################
def decode_text(code, RX_dictionary):
    """
    Inputs:
    - code: A code of a text encoded by Huffman code as a string.
    - RX_dictionary: Initialized decitionary of the receiver
    Returns:
    - decoded_text: a string represents the decoded text.
    - RX_dictionary: the updated dictionary of the receiver.
    """
    decoded_text = ''
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    while (len(code) > 0):
        l = len(bin(len(RX_dictionary)-1))-2
        first = code[:l]
        inv = dict((v, k) for k, v in RX_dictionary.items())
        if (len(code) == l):
            decoded_text += inv[first]
            break
        else:
            second = code[l:2*l]
            t = inv[first]+inv[second]
            decoded_text+=t
            l2 = len(RX_dictionary)
            if (l2 & (l2-1) == 0):
                for i in RX_dictionary.keys():
                    RX_dictionary[i] = "0"+RX_dictionary[i]
            RX_dictionary[t] = bin(l2)[2:]
        code = code[l*2:]
```

```
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return decoded_text, RX_dictionary

decoded_text,RX_dictionary = decode_text(code, RX_dictionary)
print("Orginal text:   ", text)
print("Decoded Text:   ", decoded_text)
print("The receiver Dictionary:   ", RX_dictionary)
```

```
Orginal text:    AAAABBBBBBAAAAACCCCCEEEEAAAADDDDBBBAAACCCDDDEEEEEBBBCCCC
Decoded Text:    AAAABBBBBBAAAAACCCCCEEEEAAAADDDDBBBAAACCCDDDEEEEEBBBCCCC
The receiver Dictionary:    {'D': '00000', 'E': '00001', 'B': '00010', 'C':
'00011', 'A': '00100', 'AA': '00101', 'AAB': '00110', 'BB': '00111', 'BBB':
'01000', 'AAA': '01001', 'AAC': '01010', 'CC': '01011', 'CCE': '01100', 'EE':
'01101', 'EA': '01110', 'AAAD': '01111', 'DD': '10000', 'DB': '10001', 'BBA':
'10010', 'AACC': '10011', 'CD': '10100', 'DDE': '10101', 'EEE': '10110', 'EB':
'10111', 'BBC': '11000', 'CCC': '11001'}
```

[ ]: