ECE 231A Project 1

Lawrence Liu, David Zheng, Rohit Bhat

November 14, 2022

Data_Compression_Project_Huffman

November 14, 2022

0.1 ECE 231A : Data Compression Project Module 1

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

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import csv
from collections import Counter
```

[3]:
```
#############################################################################################
# Read Text file
#############################################################################################
f = open("Toy_Example_Huffman.txt", "r")
text = f.read()
```

[25]:
```
#############################################################################################
# Compute the empirical 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.
    """
    # ================================================================== #
    # YOUR CODE HERE:
    # ================================================================== #
    chars, counts = np.unique(list(text), return_counts=True)
    symbols = [(str(char), count/len(text)) for char, count in zip(chars,␣
↪counts)]
    # ================================================================== #
    # END YOUR CODE HERE
```

```
    # ================================================================ #
    return symbols

symbols = compute_distribution(text)
size_symbols = len(symbols)
print(symbols)
```

```
[('A', 0.125), ('B', 0.125), ('C', 0.25), ('D', 0.5)]
```

0.2 Part 1: D-ary Huffman Codes

[26]:
```
##########################################################################################
# Draw the tree for the Huffman code
##########################################################################################
def Huffman_tree(symbols, D = 3):
    """
    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:
    - tree: a list of a single element that have probability one. at each␣
  ↪iteration sort your list according
              to their probabilities and combine the first D elements as a single␣
  ↪element
    """
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    #check if we need to add dummy symbols

    if len(symbols)==1:
        return symbols
    if (len(symbols)-1)%(D-1)>0:
        for i in range((D-1)-((len(symbols)-1)%(D-1))):
            symbols.append(('dummy'+str(i),0))
    char,prob=zip(*symbols)
    char=list(char)
    prob=list(prob)

    #sort the chars and probs
    sorted_chars=[x for _,x in sorted(zip(prob,char),key=lambda pair: pair[0])]
    sorted_probs=sorted(prob)
    #combine the first D elements as a single element
    tree=[(tuple(sorted_chars[:D]),sum(sorted_probs[:D]))]
```

```python
    #add the rest of the elements
    for i in range(D,len(sorted_chars)):
        tree.append((sorted_chars[i],sorted_probs[i]))
    return Huffman_tree(tree,D)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return tree

tree = Huffman_tree(symbols)
print(tree[0][0])
```

```
(('dummy0', 'A', 'B'), 'C', 'D')
```

[28]:
```python
########################################################################################
#Encode the Huffman Tree
########################################################################################
def Huffman_coding(seq, code='', D =3):
    """
    Inputs:
    - seq: a tuple of characters.
    - code: the code of this tuple
    Returns:
    - Dictionary: a dictionary containing the Huffman codes.
    """
    if type(seq) is str:
        return {seq : code}
    Dictionary = dict()
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    dicts=[{}]*D
    for i in range(D):
        dicts[i]=Huffman_coding(seq[i],code+str(i),D)
    for d in dicts:
        Dictionary.update(d)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return Dictionary

Huffman_code = Huffman_coding(tree[0][0])
print("Huffman CodeBook:  ", Huffman_code)
```

```
Huffman CodeBook:   {'dummy0': '00', 'A': '01', 'B': '02', 'C': '1', 'D': '2'}
```

[29]:
```python
####################################################################################
# Compute the expected length of the Huffman code
####################################################################################
def compute_expected_length(symbols, Huffman_code, D =3):
    """
    Inputs:
    - symbols: A list of tuples of the form (char,prob).
    - Huffman_code: a dictionary containing the Huffman codes.Each code is a␣
 ↪string
    Returns:
    - Expected_length: a number represents the expected length of Huffman code
    """

    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    Expected_length=0
    for symbol in symbols:
        char=symbol[0]
        Expected_length+=symbol[1]*len(Huffman_code[char])

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return Expected_length

Expected_length = compute_expected_length(symbols, Huffman_code)
print("Expected length of Huffman code:   ", Expected_length)
```

```
Expected length of Huffman code:    1.25
```

[30]:
```python
####################################################################################
# Encode a text
####################################################################################
def encode_text(text, Huffman_code, D =3):
    """
    Inputs:
    - text: A string containing the text to be encoded.
    - Huffman_code: a dictionary containing the Huffman codes.Each code is a␣
 ↪string
    Returns:
    - txt_code: a string represents the code of the input text.
    """
    txt_code = ''
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
```

```
    for char in text:
        txt_code+=Huffman_code[char]
    # ================================================================= #
    # END YOUR CODE HERE
    # ================================================================= #
    return txt_code
txt_code = encode_text(text, Huffman_code)
print("Encoded Text:  ", txt_code)
```

```
Encoded Text:   2011220212
```

[31]:
```
####################################################################################
# Decode a text
####################################################################################
def decode_text(txt_code, Huffman_code, symbols, D =3):
    """
    Inputs:
    -symbols: a list of symbols.
    - txt_code: A code of a text encoded by Huffman code as a string.
    - Huffman_code: a dictionary containing the Huffman codes. Each code is a␣
 ↪string
    Returns:
    - decoded_text: a string represents the decoded text.
    """
    decoded_text = ''
    # ================================================================= #
    # YOUR CODE HERE:
    # ================================================================= #
    r_Huffman_code={v:k for k,v in Huffman_code.items()}
    codeword=''
    for char in txt_code:
        codeword+=char
        if codeword in r_Huffman_code:
            decoded_text+=r_Huffman_code[codeword]
            codeword=''

    # ================================================================= #
    # END YOUR CODE HERE
    # ================================================================= #
    return decoded_text

decoded_text = decode_text(txt_code, Huffman_code, symbols)
print("Orginal text:   ", text)
print("Decoded Text:   ", decoded_text)
```

```
Orginal text:    DACDDBCD
Decoded Text:    DACDDBCD
```

```
[ ]:
```

ECE 231A Project 3 Part 2

November 14, 2022

Problem 1

We can rewrite $P(l(x) < l'(x)) > P(l(x) > l'(x))$ as

$$\sum_x p(x) sign(l(x) - l'(x)) \leq 0$$

using the identity $sign(a - b) \leq D^t - 1$, we get

$$\begin{aligned} \sum_x p(x) sign(l(x) - l'(x)) &\leq \sum_x p(x) \left(D^{l(x) - l'(x)} + 1 \right) \\ &= \sum_x D^{l'(x)} - 1 \end{aligned}$$

From the kraft inequality, we then get that $\sum_x D^{l'(x)} \leq 1$. and thus we get that $\sum_x D^{l'(x)} - 1 \leq 0$.

Problem 2

(a)

let $l'(x) = -\log(q(x))$, then we have

$$\begin{aligned}
p(l'(x) \leq l^*(x) - \gamma) &= p(-\log_D(q(x)) \leq -\log_D(p(x)) - \gamma) \\
&= p(q(x) \geq p(x)D^\gamma) \\
&= \sum_{x:p(x)D^\gamma \leq q(x)} p(x) \\
&\leq \sum_{x:p(x)D^\gamma \leq q(x)} q(x)D^{-\gamma} \\
&= \sum_x q(x)D^{-\gamma} \\
&= D^{-\gamma}
\end{aligned}$$

Thus we get

$$p(l'(x) \leq l^*(x) - \gamma) \leq |D|^{-\gamma}$$

(b)

We have

$$\begin{aligned}
p(l(x) > l'(x) + 1) &= p\left(\lceil \log_D\left(\frac{1}{p(x)}\right)\rceil > l'(x) + 1\right) \\
&\leq p(1 - log_D(p(x)) > l'(x) + 1) \\
&=\leq p(-\log_D(p(x)) < l'(x))
\end{aligned}$$

Since $l(x)$ must be an integer we have that this is equal to

$$p(-\log_D(p(x)) < l'(x)) = p(l'(x) \leq -\log_D(p(x)) - 1)$$

from part (a) we know that

$$p(l'(x) \leq -\log_D(p(x)) - 1) \leq D^{-1} \leq \frac{1}{2}$$

since $D \geq 2$. thus we have that

$$p(l(x) > l'(x) + 1) \leq \frac{1}{2}$$

and thus

$$p(l(x) > l'(x) + 1) \leq p(l(x) \leq l'(x) + 1)$$

(c)

$$P[l'(x) < l(x) - 1] < P[l'(x) > l(x) - 1]$$

$$\mathbb{E}[l'(x)] \leq H(x) + 1$$

$$\mathbb{E}[l(x) - 1] \leq H(x) \leq \mathbb{E}[l'(x)]$$

$$\sum_{i \in X-1} \left[p(x_i) \log \frac{1}{p(x_i)} \right] < \sum_{i \in X} p(x_i) \log \frac{1}{p(x_i)} \leq \sum_{i \in X} p(x_i) \log \left(\frac{1}{p(x_i)} \right)$$

$$\mathbb{P}[l(x) - 1 > l'(x)] < \mathbb{P}[l'(x) \geq H(x) > l(x) - 1]$$

$$\mathbb{P}[l(x) - 1 > l'(x)] < \mathbb{P}[l'(x) > l(x) - 1] \quad \square$$

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:
```

```python
        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]:

```python
###########################################################################################
# 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 10010010101001001011101001010100010100110011001110110001
000100010001010010010000000000000000000001000111001000101000011000110000010000000
010110100001000010001000111000110101100011
```

[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'}
```

[ ]:

```

```

Data_Compression_Project_Arithmetic_1

November 14, 2022

[20]:
```python
import numpy as np
import matplotlib.pyplot as plt
import csv
from collections import Counter
```

[21]:
```python
##########################################################################################
# Read Text file
##########################################################################################
f = open("Toy_Example_Arithmetic.txt", "r")
text = f.read()
print(text)
```

```
DACDDBCD
```

[22]:
```python
##########################################################################################
# Compute the empirical 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.
    """
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    symbols = []
    counter = dict()
    for k in text:
        if k in counter:
            counter[k]+=1/len(text)
        else:
            counter[k]=1/len(text)
```

```
    symbols = []
    for j in counter.keys():
        symbols.append((j, counter[j]))
    symbols = sorted(symbols, key=lambda x:x[1])
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return symbols

symbols = compute_distribution(text)
size_symbols = len(symbols)
print(symbols)
```

[('A', 0.125), ('B', 0.125), ('C', 0.25), ('D', 0.5)]

0.1 Part 2: Arithmetic Codes

[23]:
```
#######################################################################################
# Compute the expected length of the Huffman code
#######################################################################################
def compute_CDF(symbols):
    """
    Inputs:
    - symbols: A list of tuples of the form (char,prob).
    Returns:
    - CDF_symbols: A list of tuples of the form (char,CDF)
    """
    CDF_symbols = []
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    total=0
    for i in symbols:
        CDF_symbols.append((i[0], i[1]+total))
        total += i[1]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return CDF_symbols

CDF_symbols = compute_CDF(symbols)
print(CDF_symbols)
```

[('A', 0.125), ('B', 0.25), ('C', 0.5), ('D', 1.0)]

[24]:
```
###########################################################################
# Decimal encoding
```

```
############################################################################
def decimal_encoding(text,CDF_symbols) :
    """
    Inputs:
    - text: A string containing the text to be encoded.
    Returns:
    - lower: the lower value of the interval of the encoded text.
    - upper: the upper value of the interval of the encoded text.
    """
    lower = 0
    upper = 1
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    d = dict(CDF_symbols)
    sy = [i[0] for i in CDF_symbols]
    for c in text:
        r = upper - lower
        upcdf = d[c]
        ind = sy.index(c)
        locdf = 0
        if (ind != 0):
            locdf = d[sy[ind-1]]
        lower += r* locdf
        upper -= r*(1-upcdf)



    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return lower, upper

lower,upper = decimal_encoding(text,CDF_symbols)
print("Interval representing the text is:  ", lower, upper)
```

```
Interval representing the text is:   0.52801513671875 0.528076171875
```

[25]:

```
############################################################################
# Binary encoding
############################################################################
def Arithmetic_encoding(lower,upper):
    """
    Inputs:
    - lower: the lower value of the interval of the encoded text.
    - upper: the upper value of the interval of the encoded text.
    Returns:
```

```
    - txt_code: a string represents the code of the input text.
    """
    txt_code = ''
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    l = np.ceil(np.log2(1/(upper -lower)))+1
    m = (upper + lower)/2
    m *= (2**l)
    m = int(m)
    return bin(m)[2:]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return txt_code

txt_code = Arithmetic_encoding(lower,upper)
print("Encoded Text:  ", txt_code)
Expected_length_Arithematic = len(txt_code)/len(text)
print("Expected length of Arithematic code:   ", Expected_length_Arithematic)
```

```
Encoded Text:   100001110010111
Expected length of Arithematic code:    1.875
```

[26]:
```
###########################################################################
# Binary Decoding
###########################################################################
def decimal_decoding(txt_code):
    """
    Inputs:
    - txt_code: a string of zeros and ones represents the code of the input␣
 ↪text.
    Returns:
    - decoded_val: a real number between 0 and 1.
    """
    decoded_val = 0
    # ================================================================ #
    # YOUR CODE HERE:
    # ================================================================ #
    decoded_val =  int(txt_code, 2) / (2 ** len(txt_code))
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return decoded_val

decoded_val = decimal_decoding(txt_code)
print("The decoded Value:  ", decoded_val)
```

```
The decoded Value:   0.528045654296875
```

[27]:

```
######################################################################
# Arithmetic Decoding
######################################################################
def Arithmetic_decode(decoded_val,CDF_symbols, n):
    """
    Inputs:
    - decoded_val: A real number between 0 and 1 represents the mid-point of␣
 ↪the interval of the encoded text.
    - CDF_symbols: A list containing the symbols and their corresponding CDF
    - n: number of symbols to be decoded
    Returns:
    - decoded_text: a string containing the decoded text.
    """
    decoded_text = ''
    # ============================================================== #
    # YOUR CODE HERE:
    # ============================================================== #
    sy = [i[0] for i in CDF_symbols]
    for i in range(n):
        j = 0
        while (CDF_symbols[j][1]<decoded_val):
            j+=1
        decoded_text += sy[j]
        l = 0
        if (j != 0):
            l = CDF_symbols[j-1][1]
        decoded_val = (decoded_val - l)/(CDF_symbols[j][1] - l)


    # ============================================================== #
    # END YOUR CODE HERE
    # ============================================================== #
    return decoded_text

decoded_text = Arithmetic_decode(decoded_val,CDF_symbols, len(text))
print("Orginal text:   ", text)
print("Decoded Text:   ", decoded_text)
```

```
Orginal text:    DACDDBCD
Decoded Text:    DACDDBCD
```

[ ]:

Problem 5

5:

(a) $\log\left(\frac{P(x|\hat{\theta})}{Q(x)}\right) = \underbrace{-\log(Q(x))}_{\substack{\text{code length} \\ \text{using } Q}} - \Big(\underbrace{-\log(P(x|\hat{\Theta}))}_{\substack{\text{Code length} \\ \text{using } \hat{\Theta}}}\Big)$

This is the distance (difference) in code length between using the arbitrary distribution Q, and the max likelihood $\hat{\theta}$.

(b) Assume there exists $Q(x) \neq Q^*(x)$ such that:

$$\max_x \log\left(\frac{P(x|\hat{\Theta})}{Q^*(x)}\right) = \max_x \log\left(\frac{P(x|\hat{\theta})}{\frac{P(x|\hat{\theta})}{C}}\right) = \max_x \log\left(\frac{1}{\frac{1}{C}}\right)$$

$$= \log(C) > \max_x \log\left(\frac{P(x|\hat{\theta})}{Q(x)}\right)$$

We know that there must exist an x where $Q^*(x) > Q(x)$.
If $Q(x) \geq Q^*(x)$ for all x, then

$$\sum_{x \in X} Q(x) \geq \sum_{x \in X} Q^*(x) = 1$$

But $\sum_{x \in X} Q(x) = 1$ because it is a probability distribution.

So $Q(x) = Q^*(x)$.

This can't be true based on our assumption, so we know there is an x such that $Q^*(x) > Q(x)$
Since $Q^*(x)$ and $Q(x)$ are in denominator and $Q^*(x) > Q(x)$:

$$\log(C) = \log\left(\frac{P(x|\hat{\theta})}{Q^*(x)}\right) < \log\left(\frac{P(x|\theta)}{Q(x)}\right)$$

So:

$$\max_x \log\left(\frac{P(x|\hat{\theta})}{Q(x)}\right) > \log(C)$$

Which contradicts our assumption, so $Q(x)$ must be the minimum. ∎

Contributions of Each Group Member

Lawrence Liu worked on Part 1 and Part 2 except for the bonus question David Zheng worked on Part 3 and Part 4 and Part 5 with Rohit Bhat Rohit Bhat worked on Part 5 with David Zheng and the bonus part of Part 2