Experiment No: 05

Experiment Name: A binary symmetric channel has the following noise matrix with probability, P(Y|X) = $\begin{bmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{bmatrix}$

Theory:

The channel capacity C for a Binary Symmetric Channel (BSC), we need the channel transition probability matrix, which typically provides the probabilities of bit flips (errors) in a communication system.

For a Binary Symmetric Channel, the transition probability matrix is often represented as:

$$P(x) = \begin{pmatrix} 1-p & p \\ p & 1-p \end{pmatrix}$$

Here:

- p is the probability of a bit flip (i.e., the probability of an error).
- $1 - p$ is the probability that the transmitted bit is received correctly.

The channel capacity C for a Binary Symmetric Channel is given by the formula:

$$C = 1 - H(p)$$

Where H(p) is the binary entropy function, defined as:

$$H(p) = -p \log_2 p - (1-p) \log_2 (1-p)$$

To find the channel capacity, follow these steps:

1. Identify the noise probability p from the noise matrix.
2. Calculate the binary entropy function $H(p)$.
3. Substitute H(p)H(p)H(p) into the channel capacity formula $C = 1 - H(p)$

Example

$$P(x) = \begin{pmatrix} \frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{pmatrix}$$

1. Identify the noise probability p from the noise matrix.

$$p = \frac{1}{3}$$

2. Calculate the binary entropy function $H(p)$.

$$H(p) = -p \log_2 p - (1-p) \log_2 (1-p)$$
$$= -\frac{1}{3} \log_2 \left(\frac{1}{3}\right) - \frac{2}{3} \log_2 \left(\frac{2}{3}\right) = 0.918 \; bits/msg \; symbol$$

3. Substitute H(p)H(p)H(p) into the channel capacity formula C

$$C = 1 - H(p) = 1 - 0.918 = \; 0.082 \; bits/msg \; symbol$$

Source Code:

```
import math

# given
matrix = [[2 / 3, 1 / 3], [1 / 3, 2 / 3]]
print("Symmetric matrix is:")
for i in range(0, 2):
    for j in range(0, 2):
        print('%.2f ' % matrix[i][j], end=' ')
    print()

# Calculate H(Y/X) using formula (1-p)log(1/(1-p))+plog(1/p)
Hp = matrix[0][0] * math.log2(1.0 / matrix[0][0]) + matrix[0][1] * math.log2(1.0 / matrix[0][1])
print("Conditional probability H(Y/X) is = %.3f" % Hp, "bits/msg symbol")

# Now calculate channel capacity using formula C = 1- H(Y/X)
C = 1 - Hp
print("Channel Capacity is = %.3f" % C, "bits/msg symbol")
```

Output:

Symmetric matrix is:

0.67 0.33
0.33 0.67

Conditional probability H(Y/X) is = 0.918 bits/msg symbol
Channel Capacity is = 0.082 bits/msg symbol

Experiment No: 06

Experiment Name: Check Optimality of Huffman Code

Theory:

To check the optimality of a Huffman code, you can use the Kraft inequality, which states that the sum of the code word lengths (in bits) raised to the power of -1 must be less than or equal to 1. If a code satisfies the Kraft inequality, it is considered optimal.

Another way to check the optimality of Huffman code is to verify that the code is prefix-free, meaning no code word is a prefix of any other code word. A prefix-free code is always optimal.

A third way to check the optimality of Huffman code is by comparing the average length of the code to the entropy of the source. In all cases, if the code is not optimal, you can use the standard algorithm for constructing a Huffman code to generate a new, optimal code.

In this source code, I am using Kraft inequality check for proving optimality of Huffman code.

Source Code:

```
import heapq
import math
from collections import Counter

def calculate_frequency(my_text):
    my_text = my_text.upper().replace(' ', '')
    frequency = dict(Counter(my_text))
    return frequency

def build_heap(freq):
    heap = [[weight, [char, ""]] for char, weight in freq.items()]
    heapq.heapify(heap)
    return heap

def build_tree(heap):
    while len(heap) > 1:
        lo = heapq.heappop(heap)
        hi = heapq.heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return heap[0]
```

```
def compute_huffman_avg_length(freq, tree, length):
    huffman_avg_length = 0
    for pair in tree[1:]:
        huffman_avg_length += (len(pair[1]) * (freq[pair[0]] / length))
    return huffman_avg_length

def entropy(freq, length):
    H = 0
    P = [fre / length for _, fre in freq.items()]
    for x in P:
        H += -(x * math.log2(x))
    return H

message = "aaabbbbbccccccddddee"
freq = calculate_frequency(message)
heap = build_heap(freq)
tree = build_tree(heap)
# tree=[20, ['D', '0'], ['B', '01'], ['E', '100'], ['A', '101'], ['C', '11']] not optimal
huffman_avg_length = compute_huffman_avg_length(freq, tree, len(message))
H = entropy(freq, len(message))
print("Huffman : %.2f bits" % huffman_avg_length)
print('Entropy : %.2f bits' % H)
if huffman_avg_length >= H:
    print("Huffman code is optimal")
else:
    print("Code is not optimal")
```

Output:

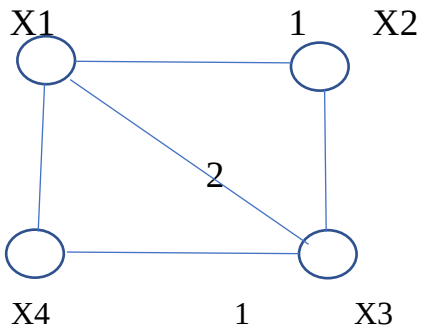
Huffman: 2.25 bits

Entropy: 2.23 bits

Huffman code is optimal

Experiment No: 07

Experiment Name: Explain entropy rate of a random walk on a weighted graph



Theory: The entropy of a stochastic process {Xi} is defined by,

$$H(X) = \lim_{n\to\infty} 1/n\, H(X_1, X_2, \ldots, X_n)$$

when the limit exists. We now consider some simple examples of stochastic processes and their corresponding entropy rates.

1. Typewriter.

Consider the case of a typewriter that has m equally likely output letters. The typewriter can produce mn sequence of length n, all of them equally likely. Hence

$H(X_1, X_2 ..., X_n) = \log \frac{m}{n}$ and the entropy rate is

$H(X) = \log m$ bits per symbol.

2. $X_1, X_2, ...$ are iid random variables. Then

$$H(X) = lim\, H(X_1, X_2, \ldots, X_n)/n = lim\, n\, H(X_1)/n = H(X_1)$$

Sequence of independent but not identically distributed random variables.

In this case,

$$H(X_1, X_2 ..., X_n) = \sum H(X_i)$$

But the $H(X_i)$'s are all not equal. We can choose a sequence of distributions on $X_1, X_2, ...$ such that the limit of $1/n\ \Sigma\ H(X_i)$ does not exist.

Consider a graph with m nodes labeled {1, 2, ..., m} with weight $W_{ij} \geq 0$ on the edge joining node i to j. (The graph is assumed to be undirected, so $W_{ij} = W_{ji}$. We set $W_{ij} = 0$ if there is no edge joining nodes i and j.) A particle walks randomly from node to node in this graph. The random walk $\{X_n\}$, $X_n \in \{1, 2, ..., m\}$, is a sequence of vertices of the graph. Given $X_n = i$, the next vertex j is chosen from among the nodes connected to node i with a probability proportional to the weight of the edge connecting i to j. Thus $P_{ij} = W_{ij} / \sum_k W_{ik}$.

In this case, the stationary distribution has a surprisingly simple form, which we will guess and verify. The stationary distribution for this Markov chain assigns probability to node i proportional to the total weight of the edges emanating from node i.

Let,

$$W_i = \sum_j W_{ij}$$

be the total weight of edges emanating from node i, and let

$$W = \sum_{i,j\ j>1} W_{ij}$$

be the sum of the weights of all the edges. Then $\sum_i W_i = 2W$. We now guess that the stationary distribution is

$$\mu_i = W_i / 2W$$

We verify that this is the stationary distribution by checking that $\mu_p = \mu$. Here,

$$\sum_i \mu_i P_{ij} = \sum_i \frac{W_i W_{ij}}{2W W_i}$$

$$= \sum_i 1/2W \, W_{ij}$$

$$= W_i / 2W$$

$$= \mu_j$$

Thus, the stationary probability of state i is proportional to the weight of edges emanating from node i. This stationary distribution has an interesting property of locality: it depends only on the total weight and the weight of edges connected to the node and hence does not change. If the weights in some other part of the graph are changed while keeping the total weight constant, we can now calculate the entropy rate as

$$H(X) = H(X_2 \mid X_1)$$

$$= - \sum_i W_i / 2W \ \sum_j W_{ij} / W_i \ \log W_{ij} / W_i$$

$$= - \sum_i \sum_j \frac{W_i}{2W} \log \frac{W_{ij}}{W_i}$$

$$= - \sum_i \sum_j \frac{W_{ij}}{2W} \log \frac{W_{ij}}{W_i} + \sum_i \sum_j \frac{W_{ij}}{2W} \log \frac{W_{ij}}{2W}$$

$$= H(\ldots\ldots \frac{W_{ij}}{2W} \ldots\ldots) - H(\ldots \frac{W_i}{2W})$$

If all the edges have equal weight, the stationary distribution puts weight E_i /2E on node i, where E_i is the number of edges emanating from node i and E is the total number of edges in the graph. In this case, the entropy rate of the random walk is

$$H(X) = \log 2E - H(E_1/2E, E_2/2E, ..., E_m/2E)$$

This answer for the entropy rate is so simple that it is almost misleading. Apparently, the entropy rate, which is the average transition entropy, depends only on the entropy of the stationary distribution and the total number of edges.

Source Code:

```
import math
from collections import defaultdict

# given
g = defaultdict(list)
xij = [[1, 1, 2], [1, 1], [1, 2, 1], [1, 1]]

def makeGraph(li):
    for node in range(len(li)):
        for x in li[node]:
            g[node].append(x)

def entropy(li):
    H = 0
    for x in li:
        if x == 0:
            continue
        H += -(x * math.log2(x))
    return H

# make graph
```

```
makeGraph(xij)
wi = []
for node in range(len(g)):
    wi.append(sum(g[node]))

# we know
# summation(wi)=2w
w = sum(wi) / 2

# the stationary distribution is
# ui=(wi)/2w
ui = [weight / (2 * w) for weight in wi]

# H((wi)/2w)=H(ui)
H_wi_div_2w = entropy(ui)

# H(wij/2*w) = H(g[]/2*w)
wij_div_2w_list = []
for i in range(len(g)):
    wij_div_2w_list += [weight / (2 * w) for weight in g[i]]

# H(wij/2*w) = H(wij_div_2w_list)
H_wij_div_2w = entropy(wij_div_2w_list)

# finally the entropy rate
# H(x)=H(wij/2w)-H(wi/2w)
H_x = H_wij_div_2w - H_wi_div_2w
print('Entropy Rate: %.2f' % H_x)
```

Output:

Entropy Rate: 1.33

Experiment No: 08

Experiment Name: Numerical Based on Conditional and Joint Entropy.

Example 2.2.1: Let (X, Y) have the following joint distribution:

X / Y	1	2	3	4
1	$1/8$	$1/16$	$1/32$	$1/32$
2	$1/16$	$1/8$	$1/32$	$1/32$
3	$1/16$	$1/16$	$1/16$	$1/16$
4	$1/4$	0	0	0

The marginal distribution of X is $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$, and the marginal distribution of Y is $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$, and hence:

H(X) = $\frac{7}{4}$ bits and H(Y) = 2 bits

Also,

$$H(X|Y) = \sum_{i=0}^{4} P(Y = i)\, H(X|Y = i)$$

$$= \frac{1}{4} H\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right) + \frac{1}{4} H\left(\frac{1}{4}, \frac{1}{2}, \frac{1}{8}, \frac{1}{8}\right) + \frac{1}{4} H\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right) + \frac{1}{4} H(1, 0, 0, 0)$$

$$= \frac{1}{4} \times \frac{7}{4} + \frac{1}{4} \times \frac{7}{4} + \frac{1}{4} \times 2 + \frac{1}{4} \times 0$$

$$= \frac{11}{8} \text{ bits}$$

Similarly, H(X|Y) = $\frac{13}{8}$ bits and H(X,Y) = $\frac{27}{8}$ bits.

Theory:

Definition: The entropy H(X) of a discrete random variable X is defined by:

$$H(X) = -\sum_{x \epsilon X} P(x) \log P(x)$$

Definition: Given (X,Y) ~ P(x,y), the conditional entropy H(Y|X) is defined as:

$$H(X|Y) = \sum_{x \epsilon X} P(x)\, H(Y|X = x)$$

$$= -\sum_{x \epsilon X} P(x) \sum_{y \epsilon Y} P(y|x) \log P(y|x)$$

$$= -\sum_{x \epsilon X} \sum_{y \epsilon Y} P(x, y) \log P(y|x)$$

$$= -E \log P(Y|X)$$

Source Code:

```
# given
import math

matrix = [
    [1 / 8, 1 / 16, 1 / 32, 1 / 32],
    [1 / 16, 1 / 8, 1 / 32, 1 / 32],
    [1 / 16, 1 / 16, 1 / 16, 1 / 16],
    [1 / 4, 0, 0, 0]
]

# the marginal distribution of x
marginal_x = []
for i in range(len(matrix[0])):
    marginal_x.append(sum(matrix[j][i] for j in range(len(matrix))))

# the marginal distribution of y
marginal_y = []
for i in range(len(matrix)):
    marginal_y.append(sum(matrix[i][j] for j in range(len(matrix[0]))))

# H(x)
def entropy(marginal_var):
    H = 0
    for x in marginal_var:
        if x == 0:
            continue
        H += -(x * math.log2(x))
    return H

H_x = entropy(marginal_x)
H_y = entropy(marginal_y)

# conditional entropy
# H(x/y)
H_xy = 0
```

```
for i in range(len(matrix)):
    tmp = [(1 / marginal_y[i]) * matrix[i][j] for j in range(len(matrix[0]))]
    H_xy += entropy(tmp) * marginal_y[i]

# H(y/x)
H_yx = 0
for i in range(len(matrix[0])):
    tmp = [(1 / marginal_x[i]) * matrix[j][i] for j in range(len(matrix))]
    H_yx += entropy(tmp) * marginal_x[i]

print('Conditional Entropy H(x|y): ', H_xy)
print('Conditional Entropy H(y|x): ', H_yx)

# Joint entropy
# H(x,y)
H_of_xy = H_x + H_yx
print('Joint Entropy H(x,y): ', H_of_xy)

# Mutual Information
# I(x,y)
I_of_xy = H_y - H_yx
print('Mutual Information: ', I_of_xy)
```

Output:

Conditional Entropy H(x|y): 1.375

Conditional Entropy H(y|x): 1.625

Joint Entropy H(x,y): 3.375

Mutual Information: 0.375