HW1

September 27, 2024

1 Homework 1 - Coding Theory

2 Naiara Alonso Montes

2.0.1 Problem 1

A a) Compute code rate

The code-rate R is calculated with the following formula $R = \frac{\log_{|U|} M}{n} = \frac{k}{n}$ where k is $\log_2 M$ and n is provided. I apply the formula:

$$R = \frac{k}{n} = \frac{\log_2 8}{7} = \frac{3}{7}$$

b) Minimun distance from the code

To calculate the minimun distance, I will use the Hamming distance. It is necessary to transform the code words into strings.

```
[ ]: import numpy as np

codewords = np.array([
    [1, 0, 0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 1, 0, 1],
    [1, 1, 0, 1, 0, 1, 1],
    [0, 0, 1, 1, 0, 1, 1],
    [1, 0, 1, 0, 1, 0, 1],
    [0, 1, 1, 1, 1, 1, 0],
    [1, 1, 1, 0, 0, 0, 0]])

hamming_weights = [np.sum(codeword) for codeword in codewords]

non_zero_weights = [weight for weight in hamming_weights if weight > 0]
min_distance = min(non_zero_weights)

print(f"The minimun distances is: {min_distance}")
```

```
The minimun distances is: 3
```

c) Check if code is linear

For a code to be linear, the result of the XOR of 2 codewords must be another codeword.

```
[ ]: from os import name
     def xor(word1, word2):
         result = ""
         for i in range(len(word1)):
             if word1[i] == word2[i]:
                 result += "0"
             else:
                 result += "1"
         return result

     def main():
       for i, word1 in enumerate(codewords):
         for j, word2 in enumerate(codewords):
           resulted_word = xor(word1, word2)
           if resulted_word not in codewords:
             print(f"The code is not linear")
             exit()
       print(f"The code is linear")

     main()
```

```
The code is linear
```

B a) Parity-check matrix First I need to obtain the Identity Matrix, which is not trivial. Row 2 = Row 2 - Row 1

$$M' = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{1}$$

In matrix M' the identity matrix I_3 has dimensions 3×3 and parity matrix P has dimesions 3×4. I need the transpose matrix P'.

$$P' = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \tag{2}$$

Now the parity-check matrix $H = [P'|I_3]$, which is:

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

b) Code rate

Code rate R is defined as in a generator matrix as $\frac{k}{n}$ where k is equal to the number of rows of the matrix and n is the number of columns. For our example:

$$R = \frac{k}{n} = \frac{3}{7} \approx 0.429$$

c) Minimun distance of the code

```
[ ]: import numpy as np

generator_matrix = np.array([
    [1, 0, 0, 1, 0, 1, 1],
    [0, 1, 1, 0, 1, 1, 0],
    [0, 0, 1, 1, 0, 0, 1]
])

all_binary_vectors = np.array([
    [0, 0, 0],
    [0, 0, 1],
    [0, 1, 0],
    [0, 1, 1],
    [1, 0, 0],
    [1, 0, 1],
    [1, 1, 0],
    [1, 1, 1]
])

codewords = []
for vector in all_binary_vectors:
    codeword = np.dot(vector, generator_matrix) % 2
    codewords.append(codeword)

hamming_weights = [np.sum(codeword) for codeword in codewords]

non_zero_weights = [weight for weight in hamming_weights if weight > 0]
min_distance = min(non_zero_weights)

print(f"The minimun distance is: {min_distance}")
```

```
The minimun distance is: 3
```

d) Check minimun distance using parity check

To check the minimun distance using the parity check matrix, we need to sum all combinations of columns and see if the result is 0 in module 2. The number of 0 vectors is the minimun distance. The sum can be done with 2 or more columns.

```
[ ]: from itertools import combinations

     H = np.array([
         [1, 1, 1, 1, 0, 0, 0],
         [0, 1, 0, 0, 1, 0, 0],
         [1, 1, 0, 0, 0, 1, 0],
         [1, 1, 1, 0, 0, 0, 1]
     ])

     n = H.shape[1]

     def is_linearly_dependent(cols):
         col_sum = np.sum(cols, axis=1) % 2
         return np.array_equal(col_sum, np.zeros(cols.shape[0]))

     min_distance = n
     for r in range(2, n + 1):
         for comb in combinations(range(n), r):
             selected_columns = H[:, comb]
             if is_linearly_dependent(selected_columns):
                 min_distance = min(min_distance, r)
                 break

     print(f"The minimun distance is: {min_distance}")
```

```
The minimun distance is: 3
```

e) Decision region for 0-codeword

To get the decision region for 0-codeword we need to compare it with all possible combinations of codewords. To do so, first we need to obtain all possible combinations and calculate the distance to 0-codeword.

```
[ ]: import numpy as np
     import itertools
     import pandas as pd

     def binary_combinations(n):
       return np.array(list(itertools.product([0, 1], repeat=n)))

     def hamming_distances():
       n_combinations = binary_combinations(7)
       codeword_0 = np.array([0] * n_combinations.shape[1])
       distances = np.sum(np.abs(n_combinations - codeword_0), axis=1)
       return distances

     dataframe = pd.DataFrame({'Combination': binary_combinations(7).tolist(),␣
      ↪'Distance': hamming_distances()})
```

```
dataframe
```

[ ]:

```
                 Combination  Distance
0    [0, 0, 0, 0, 0, 0, 0]         0
1    [0, 0, 0, 0, 0, 0, 1]         1
2    [0, 0, 0, 0, 0, 1, 0]         1
3    [0, 0, 0, 0, 0, 1, 1]         2
4    [0, 0, 0, 0, 1, 0, 0]         1
..                     …       …
123  [1, 1, 1, 1, 0, 1, 1]         6
124  [1, 1, 1, 1, 1, 0, 0]         5
125  [1, 1, 1, 1, 1, 0, 1]         6
126  [1, 1, 1, 1, 1, 1, 0]         6
127  [1, 1, 1, 1, 1, 1, 1]         7

[128 rows x 2 columns]
```

We know that there exists a theorem that can correct $\lfloor (d-1)/2 \rfloor$ errors. In our case, $d = 3$, so the number of corrected errors is $\lfloor (3-1)/2 \rfloor = 1$. It means that for any vector which the distance is 1 or less, it belongs to the 0-codeword decision region.

[ ]:

```python
def decision_region(distances):
  decision_region_indexes = []
  for i, distance in enumerate(distances):
    if distance <= 1:
      decision_region_indexes.append(i)
  return decision_region_indexes

decision_region(hamming_distances())

dataframe.iloc[decision_region(hamming_distances())]
```

[ ]:

```
               Combination  Distance
0   [0, 0, 0, 0, 0, 0, 0]         0
1   [0, 0, 0, 0, 0, 0, 1]         1
2   [0, 0, 0, 0, 0, 1, 0]         1
4   [0, 0, 0, 0, 1, 0, 0]         1
8   [0, 0, 0, 1, 0, 0, 0]         1
16  [0, 0, 1, 0, 0, 0, 0]         1
32  [0, 1, 0, 0, 0, 0, 0]         1
64  [1, 0, 0, 0, 0, 0, 0]         1
```

C

2.1 Problem 2

A a) Probability of excatly e errors

The probability of getting getting exactly e errors can be explaines as follow: - We the number of bits that are going to change can be defined as $\binom{n}{e}$. - The probability of this bits to change and not the others is $p^e(1-p)^{n-e}$. So, the probability of getting exactly e errors is:

$$\binom{n}{k} \cdot p^e(1-p)^{n-e}$$

b) Probability of at least e errors

The probability of getting at least e erros is the sum of the probabilities of getting e errors or more.

$$P_{\text{e or more errors}} = \sum_{i=e}^{n} \binom{n}{i} \cdot p^i(1-p)^{n-i}$$

c) Decoding error probability of the ML decoder

The repetition code can correct $\lfloor (n-1)/2 \rfloor$ errors, so if number of errors is greater, the ML decoder will choose the wrong codeword. The probability of error in the ML decoder is the sum of probabilities that lead to that error. So:

$$P_{\text{e in ML decoder}} = \sum_{i=\frac{n+1}{2}}^{n} \binom{n}{i} \cdot p^i(1-p)^{n-i}$$

B a) Number of corrected erasures is granted?

As we are dealing with a repetition code, the erasures can be easily detected and corrected by considering that there are no errors in the received codeword. Considering that, the number of corrected erasures is based on the distances, so:

$$\text{Corrected erasures} = d - 1$$

With d being equal to the distance of the codewords.

C a) Correcting both errors and erasures probability The formula $2t + \rho \leq d - 1$ ensures that there will be enough non-erased bits to correct errors.

All possible combinations for errors and erasures is given by the bynom $\binom{n}{i+j}$ where i is the number of errors and j the number of erasures.

The probabilities of i errors and j erasures is given by $p^i \cdot \epsilon^j \cdot (1-p-\epsilon)^{n-i-j}$ where p is the error probability, ϵ the erasure probability and $(1-p-\epsilon)^{n-i-j}$ the probability of error in the remaining bits

If we put it all together, we obtain the following formula:

$$P_{\text{error}} = \sum_{i=t}^{n-\rho} \sum_{j=0}^{\rho} \binom{n}{i+j} \cdot p^i \cdot \epsilon^j \cdot (1-p-\epsilon)^{n-i-j}$$

Where: - n is the codeword length - p is the error probability of the BCS - ϵ is the erasure probability of the BSC - t is the number of errors to be corrected - $\binom{n}{i+j}$ represents the number of ways to choose $i+j$ errors or erasures out of n bits - $p^i \cdot \epsilon^j \cdot (1-p-\epsilon)^{n-i-j}$ is the probability of i errors and j erasures ocurring

2.2 Problem 3

A a) Error probability in BSC

For case 1: transmitted bit is 0 - The received signal is $r = -\sqrt{E} + n$ - An error occurs is $r > 0$, which means $n > \sqrt{E}$ - The probability of error is $P(error|s0) = Q(\sqrt{\frac{E}{\sigma 2}})$

For case 1: transmitted bit is 1 - The received signal is $r = \sqrt{E} + n$ - An error occurs is $r > 0$, which means $n > -\sqrt{E}$ - The probability of error is $P(error|s1) = Q(\sqrt{\frac{E}{\sigma 2}})$

Since two cases are symetric:

$$P(error) = Q(\sqrt{\frac{E}{\sigma^2}})$$

b) Transition probability matrix

The transition probability matrix P is:

$$P = \begin{pmatrix} P(0|0) & P(0|1) \\ P(1|0) & P(1|1) \end{pmatrix} \tag{4}$$

Since the channels are symetric:

$$P = \begin{pmatrix} 1-P(error) & P(error) \\ P(error) & 1-P(error) \end{pmatrix} \tag{5}$$

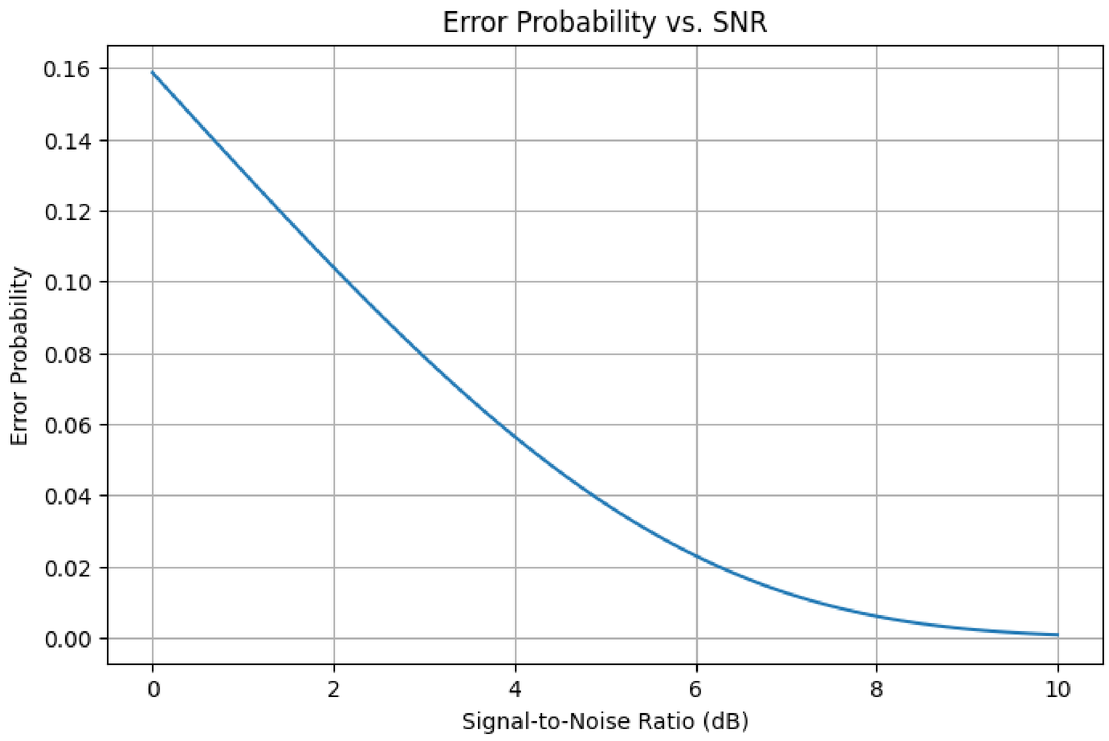
c) Transition diagram

I found the exact same transition diagram in here.

B a) Plot error probability

```
[ ]: import numpy as np
     import scipy.stats as stats
     import matplotlib.pyplot as plt

     # Define the signal-to-noise ratio range
     snr_db = np.linspace(0, 10, 100)
     snr = 10**(snr_db/10)

     # Calculate error probability
     p_error = stats.norm.sf(np.sqrt(snr))

     # Plot error probability vs. SNR
```

```python
plt.figure(figsize=(8, 5))
plt.plot(snr_db, p_error)
plt.xlabel("Signal-to-Noise Ratio (dB)")
plt.ylabel("Error Probability")
plt.title("Error Probability vs. SNR")
plt.grid(True)

plt.show()
```



b) Expression for the capacity of the BSC with error probability p

The capacity of a BSC with error probability p is given by:

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

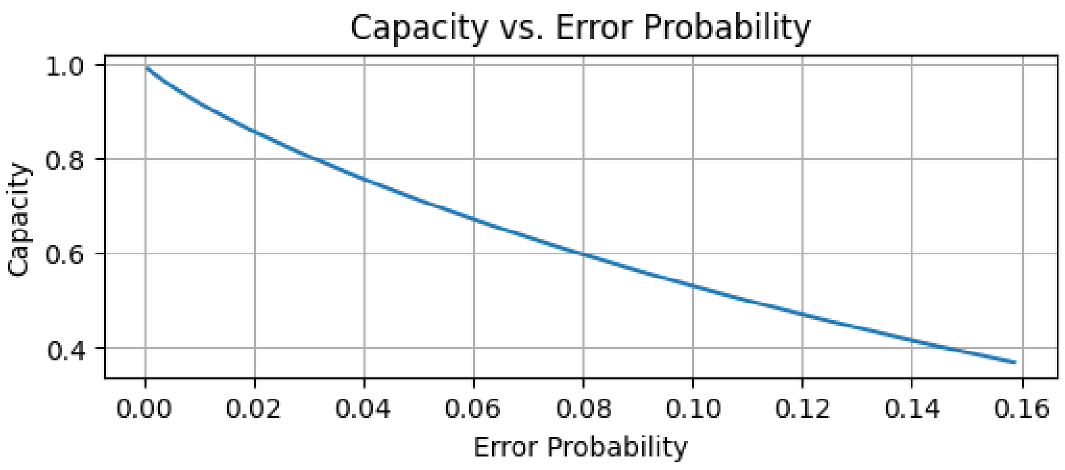
Where $H(p)$ is the binary entropy function $-p \cdot \log_2(p) - (1-p) \cdot \log_2(1-p)$.

c) Plotting capacity of BSC af funtion of p

We can plot the capacity of the BSC by calculating $P(error)$ for different values of $\frac{E}{\sigma^2}$ and the using it to calculate C. Here is the Python code:

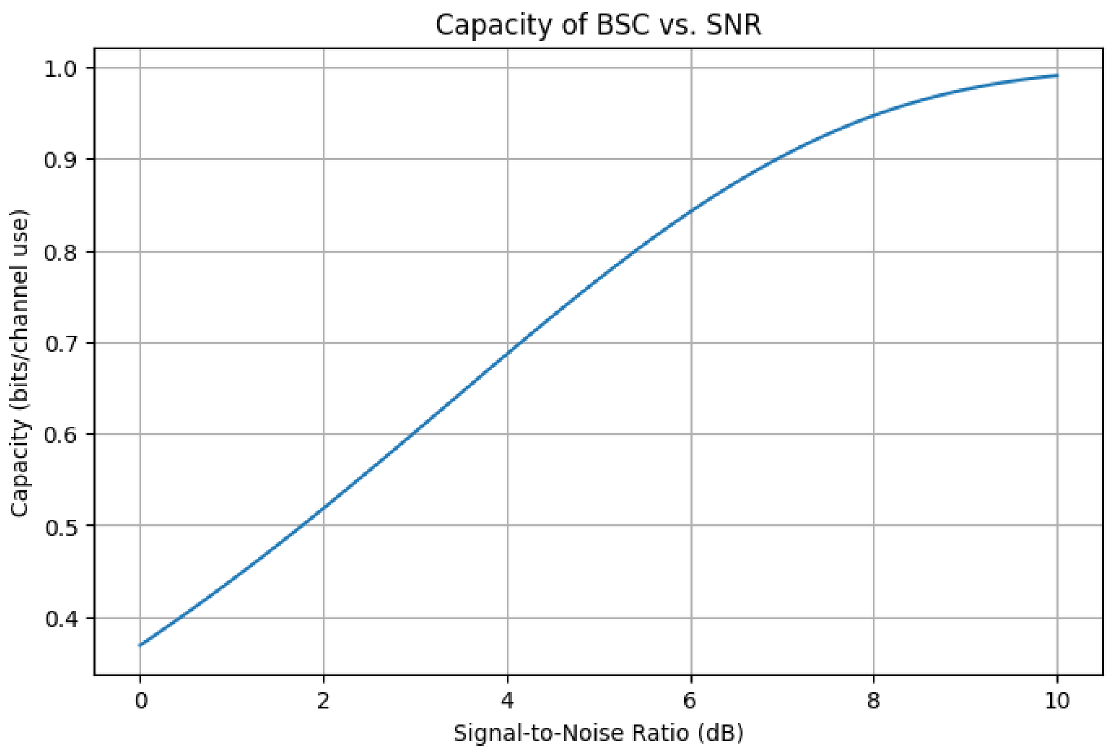
```python
[ ]: import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt
```

```
# Calculate error probability
p_error = stats.norm.sf(np.sqrt(snr))  # taking SNR from previous example

# Calculate capacity
capacity = 1 - (-p_error*np.log2(p_error) - (1-p_error)*np.log2(1-p_error))

# Plot capacity vs. p
plt.subplot(2, 1, 2)
plt.plot(p_error, capacity)
plt.xlabel("Error Probability")
plt.ylabel("Capacity")
plt.title("Capacity vs. Error Probability")

plt.grid(True)
plt.show()
```



d) Plotting capacity as SNR function

[ ]:
```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# Plot capacity vs SNR, all data as been previously computed
plt.figure(figsize=(8, 5))
plt.plot(snr_db, capacity)
plt.xlabel("Signal-to-Noise Ratio (dB)")
plt.ylabel("Capacity (bits/channel use)")
plt.title("Capacity of BSC vs. SNR")
plt.grid(True)
plt.show()
```



2.3 Problem 4

A a) Find covering radius

The covering radius is the maximun distance between all codewords. First we need to find all possible codewords and then calculate the maximun distance.

```
[ ]: import numpy as np

generator_matrix = np.array([
    [1, 0, 1, 1],
    [0, 1, 0, 1]
])

all_binary_vectors = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
```

```
codewords = []
for vector in all_binary_vectors:
    codeword = np.dot(vector, generator_matrix) % 2
    codewords.append(codeword)

hamming_weights = [np.sum(codeword) for codeword in codewords]

non_zero_weights = [weight for weight in hamming_weights if weight > 0]
max_distance = max(non_zero_weights)

print(f"The covering radius is: {max_distance}")
```

```
The covering radius is: 3
```

B a) How many correction of errors can be guaranteed?

We can correct $\lfloor \frac{(d-1)}{2} \rfloor$ errors, with d equal to the minimun distance for the code.

```
[ ]: import numpy as np
import math

generator_matrix = np.array([
    [1, 1, 0, 0, 1, 1],
    [0, 1, 1, 0, 1, 0],
    [1, 1, 0, 1, 0, 1]
])

all_binary_vectors = np.array([
    [0, 0, 0],
    [0, 0, 1],
    [0, 1, 0],
    [0, 1, 1],
    [1, 0, 0],
    [1, 0, 1],
    [1, 1, 0],
    [1, 1, 1]
])

codewords = []
for vector in all_binary_vectors:
    codeword = np.dot(vector, generator_matrix) % 2
    codewords.append(codeword)

hamming_weights = [np.sum(codeword) for codeword in codewords]
non_zero_weights = [weight for weight in hamming_weights if weight > 0]

min_distance = min(non_zero_weights)
```

```
print(f"The minimun distance is: {min_distance}")
max_errors = math.floor((min_distance - 1) / 2)

print(f"The maximun number of corrected errors is: {max_errors}")
```

```
The minimun distance is: 2
The maximun number of corrected errors is: 0
```

b) Maximal weight of the error pattern which can be solved by ML decoder

The value of the maximal weight of error solved by ML decoder is the same as the maximun number of corrected erros, which in this case is equal to **0**.

C a) Systematic generator and parity check matrix for code length $n = 5$

For a single parity-check code of length $n = 5$, the codeword consists of 4 information bits and 1 parity bit, leading to a $[5, 4]$ code, where $k = 4$ and $n = 5$.

The systematic generator matrix is:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \tag{6}$$

- The first k columns represent the identity matrix I_4.
- The last column adds the parity check bit to ensure that the sum is even.

The parity-check matrix is defined as $H = (-P^T I_r)$, where P^T is the transpose of P and I_r is the identity matrix of size $r \times r$ (with $r = n - k$).

$$P^T = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} \tag{7}$$

$$I_1 = \begin{pmatrix} 1 \end{pmatrix} \tag{8}$$

As we are working in binary, the negation does not affect, so:

$$H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{9}$$

D a) Systematic generator and parity check matrix for Hamming code of length $n = 8$

As wexare working with extended Hamming code, the original Hamming code has $n - 1 = 7$ bits. For the original Hamming code the matrix is:

$$H_7 = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{10}$$

An now wec extend it with 0s column am 1s row:

$$H_8 = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{11}$$

The generator matrix G is in the form $[I_k|P^T]$ where I_k is thevidentity matrix of size $k \times k$, where - $k = n - r - 1 = 8 - 3 - 1 = 4$ - P is the parity bit matrix from From previous H_7 we can extract P as the most right part of H. So:

$$P^T = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \tag{12}$$

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{13}$$

2.4 Problem 5

A

a) Construct the syndome table for the extended Hamming code of length $n = 8$

Considering that the matrix H was right, the syndrome table is the following

$$H_8 = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{14}$$

Syndrome	Error vector
0000	00000000
0001	00000001
0010	10000001
0011	10000000
0100	01000001
0101	01000000
0110	00100001
0111	00100000
1000	00010001
1001	00010000
1010	10010000
1011	00001000
1100	10000010
1101	00000100

Syndrome	Error vector
1110	00000011
1111	00000010

We can detect and correct 1-bit error and detect but not correct 2-bit errors.

b) Maximal weight of error pattern

The maximal weight of an error pattern that can be corrected by this syndrom decoder is **1**.

B

a) Probabilities computing

$$p(r|c_i) = p^d \cdot (1-p)^{8-d}$$
$$p(r|c_0) = p^5 \cdot (1-p)^3$$
$$p(r|c_1) = p^6 \cdot (1-p)^2$$
$$p(r|c_2) = p^6 \cdot (1-p)^2$$
$$p(r|c_3) = p^7 \cdot (1-p)^1$$
$$p(r|c_4) = p^6 \cdot (1-p)^2$$
$$p(r|c_5) = p^7 \cdot (1-p)^1$$
$$p(r|c_6) = p^7 \cdot (1-p)^1$$
$$p(r|c_7) = p^8 \cdot (1-p)^0$$
$$p(r|c_8) = p^4 \cdot (1-p)^4$$
$$p(r|c_9) = p^5 \cdot (1-p)^3$$
$$p(r|c_{10}) = p^5 \cdot (1-p)^3$$
$$p(r|c_{11}) = p^6 \cdot (1-p)^2$$
$$p(r|c_{12}) = p^5 \cdot (1-p)^3$$
$$p(r|c_{13}) = p^6 \cdot (1-p)^2$$
$$p(r|c_{14}) = p^6 \cdot (1-p)^2$$
$$p(r|c_{15}) = p^7 \cdot (1-p)^1$$
$$p(r|c_{16}) = p^4 \cdot (1-p)^4$$

b) Compute Hamming distances

[ ]:
```
import numpy as np

vector_r = np.array([[1, 1, 1, 1, 1, 0, 0, 0]])

codewords = np.array([
    [0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 1],
    [0, 0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 0, 1, 1, 1],
    [0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 1],
    [0, 0, 0, 0, 1, 0, 1, 0],
    [0, 0, 0, 0, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 1, 0, 0],
    [0, 0, 0, 0, 1, 1, 0, 1],
    [0, 0, 0, 0, 1, 1, 1, 0],
    [0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 0, 0, 0]
    ])

def hamming_distances():
    distances = []
    for codeword in codewords:
        distance = np.sum(np.abs(vector_r - codeword))
        distances.append(distance)
    return distances

distances = hamming_distances()

print(distances)
```

```
[6, 6, 7, 6, 7, 7, 8, 4, 5, 5, 6, 5, 6, 6, 7, 4]
```

C a) Formulate ML and MAP decoding rules

For ML:

$$c_{ML} = \arg\, max_{c \in C} P(y|c)$$

Where: - c_{ML} is the decoded message - y is the reived signal/vector - c is the candidate codeword from the set of codewords C - $P(y|c)$ is the probability of receiving y given that c was transmited.

For MAP:

$$c_{MAP} = \arg\, max_{c \in C} P(c|y)$$

Using Bayes Theorem:

$$P(c|y) = \frac{P(y|c)P(c)}{P(y)}$$

Where: - $P(c|y)$ is the probability of receiving y given that c was transmited. - $P(c)$ is the prior probability of codeword c. - $P(y)$ is the total probability of receiving y, computed as $P(y) = \sum_{c \in C} P(y|c)P(c)$, but it is constant for all codewords.

D a) Formulate equivalent ML decoding rule for BSC

$$P(y|c) = p^{d_H(y,c)}(1-p)^{n-d_H(y,c)}$$

Where: - $d_H(y,c)$ is the Hamming distance between y and c. - n is the length of the codeword - p is the probability of bit flipping during transmission

[3]:
```
!jupyter nbconvert --to html HW1.ipynb
```

```
[NbConvertApp] WARNING | pattern 'HW1.ipynb' matched no files
This application is used to convert notebook files (*.ipynb)
        to various other formats.

        WARNING: THE COMMANDLINE INTERFACE MAY CHANGE IN FUTURE RELEASES.

Options
=======
The options below are convenience aliases to configurable class-options,
as listed in the "Equivalent to" description-line of the aliases.
To see all configurable class-options for some <cmd>, use:
    <cmd> --help-all

--debug
    set log level to logging.DEBUG (maximize logging output)
    Equivalent to: [--Application.log_level=10]
--show-config
    Show the application's configuration (human-readable format)
    Equivalent to: [--Application.show_config=True]
--show-config-json
    Show the application's configuration (json format)
    Equivalent to: [--Application.show_config_json=True]
--generate-config
    generate default config file
    Equivalent to: [--JupyterApp.generate_config=True]
-y
    Answer yes to any questions instead of prompting.
    Equivalent to: [--JupyterApp.answer_yes=True]
--execute
    Execute the notebook prior to export.
    Equivalent to: [--ExecutePreprocessor.enabled=True]
--allow-errors
    Continue notebook execution even if one of the cells throws an error and
include the error message in the cell output (the default behaviour is to abort
conversion). This flag is only relevant if '--execute' was specified, too.
    Equivalent to: [--ExecutePreprocessor.allow_errors=True]
--stdin
    read a single notebook file from stdin. Write the resulting notebook with
default basename 'notebook.*'
```

```
    Equivalent to: [--NbConvertApp.from_stdin=True]
--stdout
    Write notebook output to stdout instead of files.
    Equivalent to: [--NbConvertApp.writer_class=StdoutWriter]
--inplace
    Run nbconvert in place, overwriting the existing notebook (only
            relevant when converting to notebook format)
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=]
--clear-output
    Clear output of current file and save in place,
            overwriting the existing notebook.
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=
--ClearOutputPreprocessor.enabled=True]
--no-prompt
    Exclude input and output prompts from converted document.
    Equivalent to: [--TemplateExporter.exclude_input_prompt=True
--TemplateExporter.exclude_output_prompt=True]
--no-input
    Exclude input cells and output prompts from converted document.
            This mode is ideal for generating code-free reports.
    Equivalent to: [--TemplateExporter.exclude_output_prompt=True
--TemplateExporter.exclude_input=True
--TemplateExporter.exclude_input_prompt=True]
--allow-chromium-download
    Whether to allow downloading chromium if no suitable version is found on the
system.
    Equivalent to: [--WebPDFExporter.allow_chromium_download=True]
--disable-chromium-sandbox
    Disable chromium security sandbox when converting to PDF..
    Equivalent to: [--WebPDFExporter.disable_sandbox=True]
--show-input
    Shows code input. This flag is only useful for dejavu users.
    Equivalent to: [--TemplateExporter.exclude_input=False]
--embed-images
    Embed the images as base64 dataurls in the output. This flag is only useful
for the HTML/WebPDF/Slides exports.
    Equivalent to: [--HTMLExporter.embed_images=True]
--sanitize-html
    Whether the HTML in Markdown cells and cell outputs should be sanitized..
    Equivalent to: [--HTMLExporter.sanitize_html=True]
--log-level=<Enum>
    Set the log level by value or name.
    Choices: any of [0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR',
'CRITICAL']
    Default: 30
    Equivalent to: [--Application.log_level]
```

```
--config=<Unicode>
    Full path of a config file.
    Default: ''
    Equivalent to: [--JupyterApp.config_file]
--to=<Unicode>
    The export format to be used, either one of the built-in formats
            ['asciidoc', 'custom', 'html', 'latex', 'markdown', 'notebook',
'pdf', 'python', 'qtpdf', 'qtpng', 'rst', 'script', 'slides', 'webpdf']
            or a dotted object name that represents the import path for an
            ``Exporter`` class
    Default: ''
    Equivalent to: [--NbConvertApp.export_format]
--template=<Unicode>
    Name of the template to use
    Default: ''
    Equivalent to: [--TemplateExporter.template_name]
--template-file=<Unicode>
    Name of the template file to use
    Default: None
    Equivalent to: [--TemplateExporter.template_file]
--theme=<Unicode>
    Template specific theme(e.g. the name of a JupyterLab CSS theme distributed
    as prebuilt extension for the lab template)
    Default: 'light'
    Equivalent to: [--HTMLExporter.theme]
--sanitize_html=<Bool>
    Whether the HTML in Markdown cells and cell outputs should be sanitized.This
    should be set to True by nbviewer or similar tools.
    Default: False
    Equivalent to: [--HTMLExporter.sanitize_html]
--writer=<DottedObjectName>
    Writer class used to write the
                                    results of the conversion
    Default: 'FilesWriter'
    Equivalent to: [--NbConvertApp.writer_class]
--post=<DottedOrNone>
    PostProcessor class used to write the
                                    results of the conversion
    Default: ''
    Equivalent to: [--NbConvertApp.postprocessor_class]
--output=<Unicode>
    Overwrite base name use for output files.
                Supports pattern replacements '{notebook_name}'.
    Default: '{notebook_name}'
    Equivalent to: [--NbConvertApp.output_base]
--output-dir=<Unicode>
    Directory to write output(s) to. Defaults
                              to output to the directory of each notebook.
```

```
To recover
                                  previous default behaviour (outputting to the
current
                                  working directory) use . as the flag value.
    Default: ''
    Equivalent to: [--FilesWriter.build_directory]
--reveal-prefix=<Unicode>
    The URL prefix for reveal.js (version 3.x).
            This defaults to the reveal CDN, but can be any url pointing to a
copy
            of reveal.js.
            For speaker notes to work, this must be a relative path to a local
            copy of reveal.js: e.g., "reveal.js".
            If a relative path is given, it must be a subdirectory of the
            current directory (from which the server is run).
            See the usage documentation
            (https://nbconvert.readthedocs.io/en/latest/usage.html#reveal-js-
html-slideshow)
            for more details.
    Default: ''
    Equivalent to: [--SlidesExporter.reveal_url_prefix]
--nbformat=<Enum>
    The nbformat version to write.
            Use this to downgrade notebooks.
    Choices: any of [1, 2, 3, 4]
    Default: 4
    Equivalent to: [--NotebookExporter.nbformat_version]

Examples
--------

    The simplest way to use nbconvert is

            > jupyter nbconvert mynotebook.ipynb --to html

            Options include ['asciidoc', 'custom', 'html', 'latex', 'markdown',
'notebook', 'pdf', 'python', 'qtpdf', 'qtpng', 'rst', 'script', 'slides',
'webpdf'].

            > jupyter nbconvert --to latex mynotebook.ipynb

            Both HTML and LaTeX support multiple output templates. LaTeX
includes
            'base', 'article' and 'report'.  HTML includes 'basic', 'lab' and
            'classic'. You can specify the flavor of the format used.

            > jupyter nbconvert --to html --template lab mynotebook.ipynb
            You can also pipe the output to stdout, rather than a file

            > jupyter nbconvert mynotebook.ipynb --stdout

            PDF is generated via latex

            > jupyter nbconvert mynotebook.ipynb --to pdf

            You can get (and serve) a Reveal.js-powered slideshow

            > jupyter nbconvert myslides.ipynb --to slides --post serve

            Multiple notebooks can be given at the command line in a couple of
            different ways:

            > jupyter nbconvert notebook*.ipynb
            > jupyter nbconvert notebook1.ipynb notebook2.ipynb

            or you can specify the notebooks list in a config file, containing::

                c.NbConvertApp.notebooks = ["my_notebook.ipynb"]

            > jupyter nbconvert --config mycfg.py

    To see all available configurables, use `--help-all`.
```

[ ]:
```
# to html
!jupyter nbconvert --to pdf HW1.ipynb
```