ELL710- CODING THEORY
ASSIGNMENT-1

Vansh Gupta
(2019EE10143)

Prelims

- For the linear code 1 (LC1), I take my G as [[1, 1, 1]]
- For the linear code 2 (LC2), my G is

  [[1 0 0 0 1 1 0]
  [0 1 0 0 0 1 1]
  [0 0 1 0 1 1 1]
  [0 0 0 1 1 0 1]].

  And my H is

  [[1 0 1 1 1 0 0]
  [1 1 1 0 0 1 0]
  [0 1 1 1 0 0 1]]

- For the text file, I take part of lyrics of popular song "Never Gonna Give You Up" by Rick Astley, and remove the newlines, to get a file of size roughly equal to 1KB
- The original and the decoded files, are both submitted in a zip file. The input file is 'rickroll.txt' and output files are of format 'rickroll_decode_ExpA_LB_dC.txt' where A ranges from 1 to 4, B is either 1 or 2, and C can be any of {10, 100, 200, 500, 5000}
- The code is submitted in both .py and .ipynb. However, I would recommend using the latter as it is divided into sections and can thus be read properly
    - One of the first few sections has all the major codes with self-explanatory names
- Lastly, some files were re-run, and even though I am using a seed value, there is some random element because of which the values differ by a little each time. Rest assured, those are negligible

Experiment-1

- After following the steps as listed in a python environment, the results were as follows for the % of modified characters from the original file

d	Changed Characters	Total Characters	% of characters changed
10	10	1108	0.9
100	97	1108	8.75
200	191	1108	17.24
500	404	1108	36.46
5000	1107	1108	99.91

Experiment-2 (And some explanations for 3)

- For this experiment, I followed the steps for encoding as given. For LC1 decoding, I took max of the 2 integers. For LC2, I took the first 4 bits after finding out the corresponding column of H with the help of syndrome, to find the final error sequence
- I define 3 kinds of errors here (And for experiment 3).
    - First is the number of k-length sequences which have errors.
        - For LC1, this is the number of impure sets (Neither all 0s, nor all 1s)
        - For LC2, this is a non (all) 0 syndrome
        - I count each such instance as 1, as there is no way to know if all 3 or 4 bits are erroneous or just 1
    - Second is the number of differences in binary equivalent of the text file with the decoded string
    - These two fulfil the requirement of "Errors you could detect, and you could correct" as asked in the problem statement
    - Last, is the number of character flips in the ascii version of decoded string from the original
- Error of the first type:
    - LC1:

d	Error-1
10	10
100	99
200	200
500	495
5000	3915

- LC2:

d	Error-1
10	10
100	99
200	191
500	457
5000	1674

- Error of the second type:
  - LC1:

d	Error-2
10	0
100	1
200	0
500	5
5000	941

  - LC2:

d	Error-2
10	0
100	3
200	13
500	69
5000	2995

- After following the steps as listed in a python environment, the results were as follows for the % of modified characters from the original file
  - LC1:

d	Changed Characters	Total Characters	% of characters changed
10	0	1108	0.0
100	1	1108	0.09
200	0	1108	0.0
500	5	1108	0.45
5000	676	1108	61.01

  - LC2:

d	Changed Characters	Total Characters	% of characters changed
10	0	1108	0.0
100	2	1108	0.18
200	10	1108	0.90
500	46	1108	4.15

5000	1041	1108	93.95

- Comparison between LC1 and LC2:
    - I note that both the methods detect roughly the same number of errors, which is also expected since errors are randomly distributed. When the number of errors increases, the repetition code is able to detect much more errors. This is probably because with that many errors, there is a chance of them landing in the same k-bit sequence, and xor of 2 binary bits is same as xor of their flipped versions.
    - For the second type of error, the repetition code outperforms the hamming code by a good margin. But one needs to keep in mind that it also takes 1.7x more storage as well making it unscalable
    - Accordingly, the number of character change shows a similar trend as above

Experiment-3 (Similar explanations as 3)

- For this experiment, I followed the steps for encoding as given. For LC1 decoding, I took max of the 2 integers. For LC2, I took the first 4 bits after finding out the corresponding column of H with the help of syndrome, to find the final error sequence
- I define 3 kinds of errors here (As for experiment 2).
    - First is the number of k-length sequences which have errors.
        - For LC1, this is the number of impure sets (Neither all 0s, nor all 1s)
        - For LC2, this is a non (all) 0 syndrome
        - I count each such instance as 1, as there is no way to know if all 3 or 4 bits are erroneous or just 1
    - Second is the number of differences in binary equivalent of the text file with the decoded string
    - These two fulfil the requirement of “Errors you could detect, and you could correct” as asked in the problem statement
    - Last, is the number of character flips in the ascii version of decoded string from the original
- Error of the first type:
    - LC1:

d	Error-1
10	1
100	1
200	1
500	2
5000	1

- LC2:

d	Error-1
10	1
100	2
200	2
500	2

- Error of the second type:
  - LC1:

d	Error-2
10	3
100	33
200	67
500	166
5000	1667

  - LC2:

d	Error-2
10	7
100	59
200	115
500	284

- After following the steps as listed in a python environment, the results were as follows for the % of modified characters from the original file
  - LC1:

d	Changed Characters	Total Characters	% of characters changed
10	2	1108	0.18
100	6	1108	0.54
200	11	1108	0.99
500	25	1108	2.23
5000	239	1108	21.57

  - LC2:

d	Changed Characters	Total Characters	% of characters changed

10	2	1108	0.18
100	9	1108	0.81
200	18	1108	1.62
500	41	1108	3.7

- Comparison:
    - While the blocks (of size k-bits) of error detected for the 2 codes are the same, the hamming code works much better in fixing these errors. The extremely low detection follows from the partial explanation given for the previous part- For hamming code, the xor stays unaffected if both the operands have a bit flip; For linear code, error is counted as all k bits not having the same binary number. For consecutive errors, this would be the same, but the wrong bit. This further affects the correction as now, max presence in a sequence is of the wrong bit, while such a constraint is a bit less likely in hamming code, which is evident for larger values of d

Experiment-4

- Most of the explanations are same as experiments 2 and 3
- The procedure required a random permutation. Since there is no straightforward function to find inverse of a permutation, I simply right-shifted the bits by M'//2. Finding the inverse was also easy, as I can left-shift the bits easily
    - Update: Did it, made it random using a mapping, and inverted it
- Error of the first type:
    - LC1:

d	Error-1
10	10
100	99
200	199
500	485

    - LC2:

d	Error-1

10	10
100	96
200	193
500	441

- Error of the second type:
  - LC1:

d	Error-2
10	0
100	1
200	1
500	15

  - LC2:

d	Error-2
10	0
100	8
200	12
500	92

- After following the steps as listed in a python environment, the results were as follows for the % of modified characters from the original file
  - LC1:

d	Changed Characters	Total Characters	% of characters changed
10	0	1108	0.0
100	1	1108	0.09
200	1	1108	0.09
500	15	1108	1.35

  - LC2:

d	Changed Characters	Total Characters	% of characters changed
10	0	1108	0.0
100	5	1108	0.45
200	6	1108	0.54
500	62	1108	5.59

- Comparison:
  - The comparison is a bit interesting in this case. The detection of erroneous blocks is pretty much the same
  - But the repetition code decode shows a vastly better performance than the hamming code.

- I do not fully understand why, but I reckon it has something to do with placement of the permutation and inverse.
- Once encoded, permutation and inverse does not affect the LC1 in any substantial way as there is no inter-dependence with the neighbours. Plus, by removal of succession of errors, it helps in decoding
- On the other hand, it affects the hamming code as the errors are no longer successive in nature. This in turn decreases the level of decoding by the virtue of the reasons mentioned for expts 2 and 3 with respect to xor