Ruchit Patel
CSE13S, Spring 2021
Prof. Darrell Long
21 April 2021

# Assignment 5: Hamming Codes

This writeup contains analysis of several graphs with variable factors. Though, most of them surround entropy. The main sub-topics are as follows:

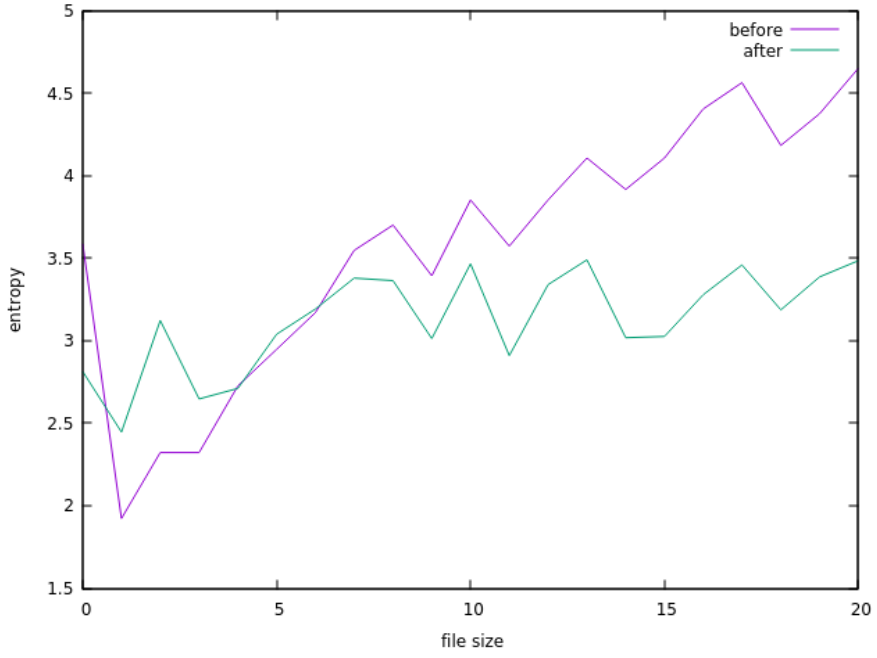
1. File Size v. Entropy
2. Error Rate v. Entropy (before encoding or decoding)
3. Error Rate v. Decode Statistics
4. Error Rate v. Entropy (after encoding and decoding)

## File Size v. Entropy

- This topic contains the analysis of entropy values before and after encoding a file.
- The file sizes are in bytes and the files contain random data produced with openssl.

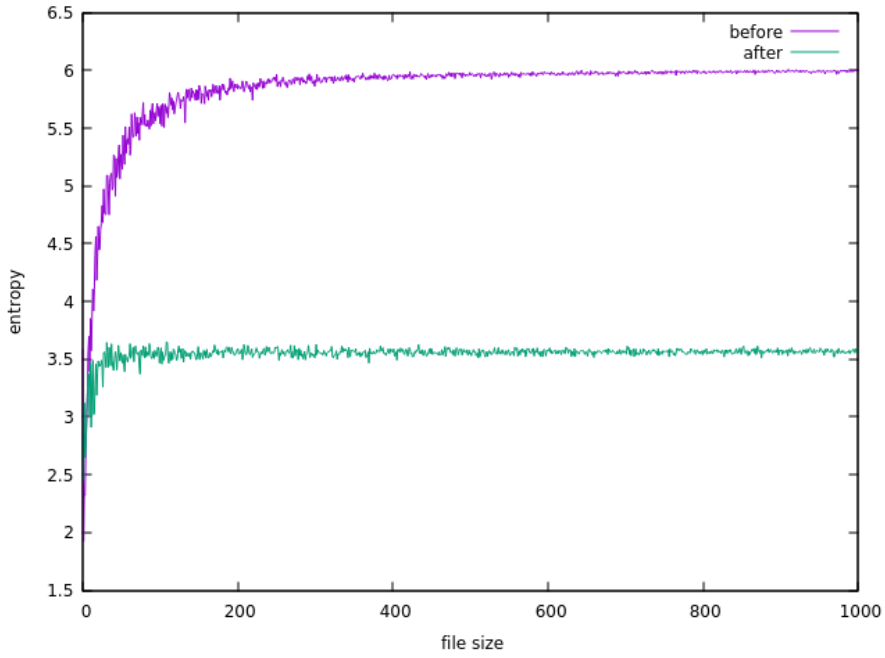
In chronological order:

1. Small file sizes
    - I started with smaller files to get an idea for a certain pattern.



- For extremely lower values, the entropy for the files before being encoded are lower at the beginning but as the file size increases a general trend where entropy of files before being encoded is higher than entropy of files after being encoded.
- From the above graph, it can be seen that the trend starts to occur at around four bytes.
- The basic reason for that is that since we are encoding both upper and lower nibbles of a byte and since there are only four bytes in the file, the information is as random as it's encoded version. Following this trend, for medium file sizes we get the following graph:

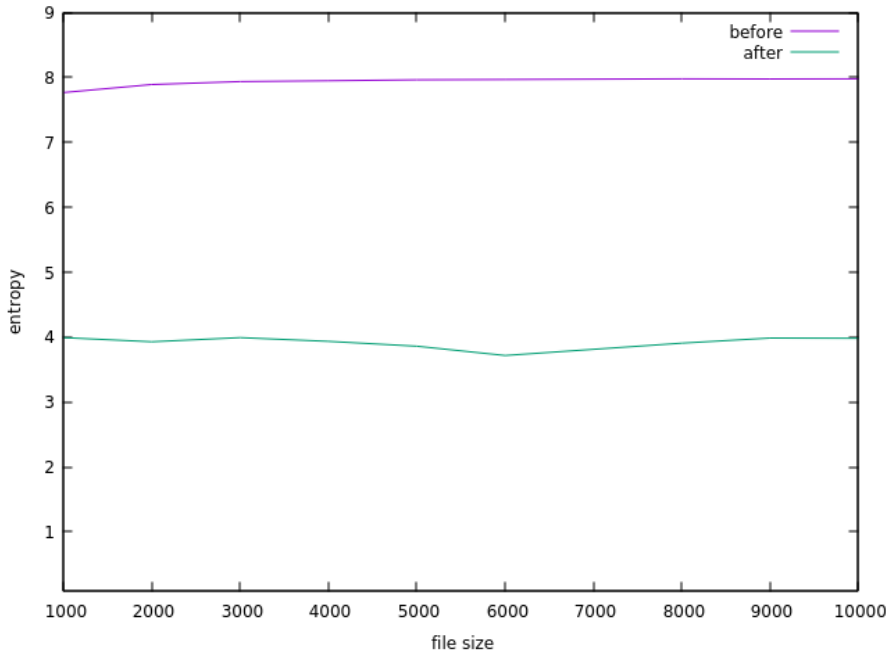
2. Medium file sizes
    - As the file size increases, it could be seen that the entropy starts to reach an horizontal asymptote.



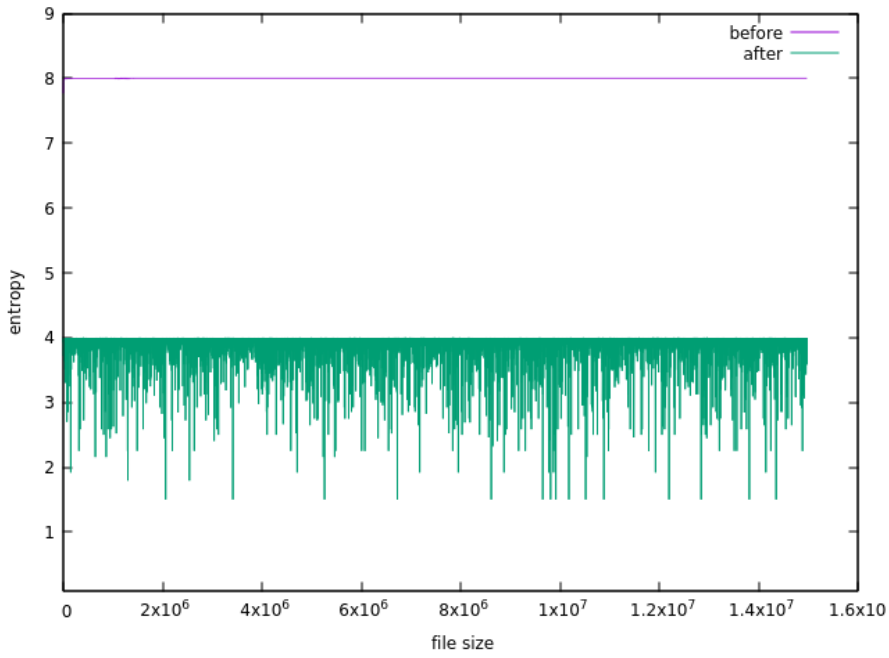

- An explanation for the above graph behavior has to do with the entropy function and the implementation of the encoder.
- The entropy function first measures the number of occurrences of each byte ranging from 0 to 255. Since, the range can be represented using 8-bits, the maximum entropy (total randomness) should be at most 8 or more formally $\log_2(256)$.
- Thus, as the file size increases, the horizontal asymptote should approach 8. Indeed that could be seen in the following topic.

- What about entropy after encoding? This is where the implementation of the encoder comes into play.
- The encoder takes in an input byte and separately encodes its upper and lower nibble. Thus, for any given byte, there could be at most $2^4$ possible outputs. Hence, similar to the previous explanation, the entropy should be 4 at max after encoding.

3. Larger file sizes
    - To see if the previous assumptions hold true, I tried even larger file sizes. The results are as following:




- From above it can be seen that the entropy of a file as the file size increases is capped at around 8 and the entropy after the file being encoded status at 4.
- Hence, the previous assumptions do hold true.
- However, the entropy for a file after being encoded did not seem as linear for even larger file sizes.



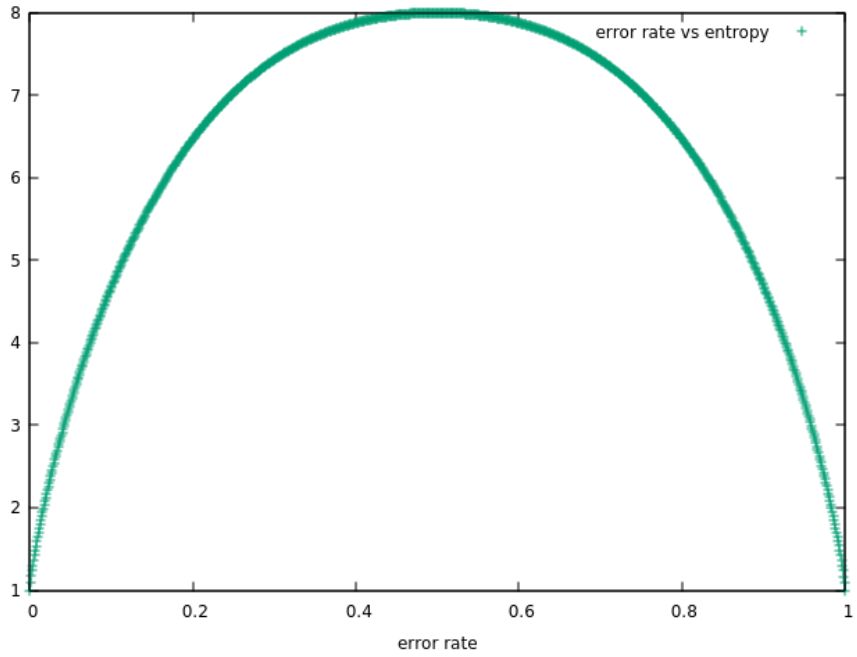

- The maximum entropy for files before and after being encoded are still 8 and 4 respectively; however, there are many downward spikes along the way.
- After researching for a while, the only explainable reason I found behind this has to do with pipes.
- All of the downward spikes started to appear after around 70,000 bytes and in the manual page of pipe it says that maximum pipe capacity is around 65,536 bytes (which also holds true for my system).
- After checking the return values for the pipe writes, I discovered that it was indeed returning an error. Hence, that is the only reason, at least that I know of, which is causing these entropy differences.
- Note that there are no such errors for the entropy before encoding a file. That is because there is no use of pipes in that case. I ran a modified version of the entropy code which reads in a file with a given buffer limit (no pipe errors).

**Error Rate v. Entropy (before encoding or decoding)**

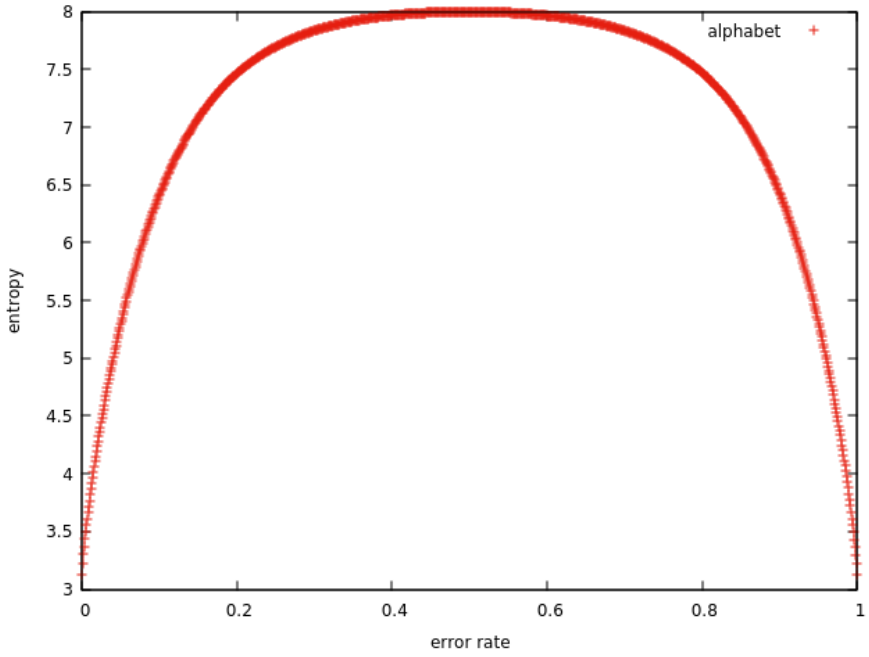
- I wanted to see what happens to the entropy of a file as the error rate increases.
- The files used are from the *Corpora* directory provided in the resources folder. From lowest to highest entropy, the files included are: *aaa*, *alphabet*, *random*.
- The files are of the same size, namely 200,000 bytes.

Graphs (analysis is after the last graph):

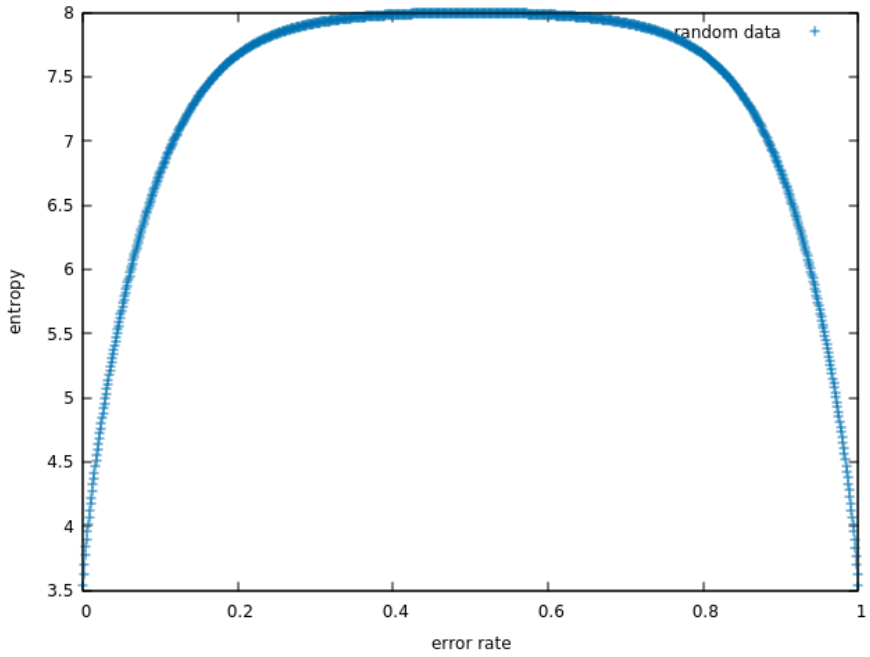
1. *aaa*



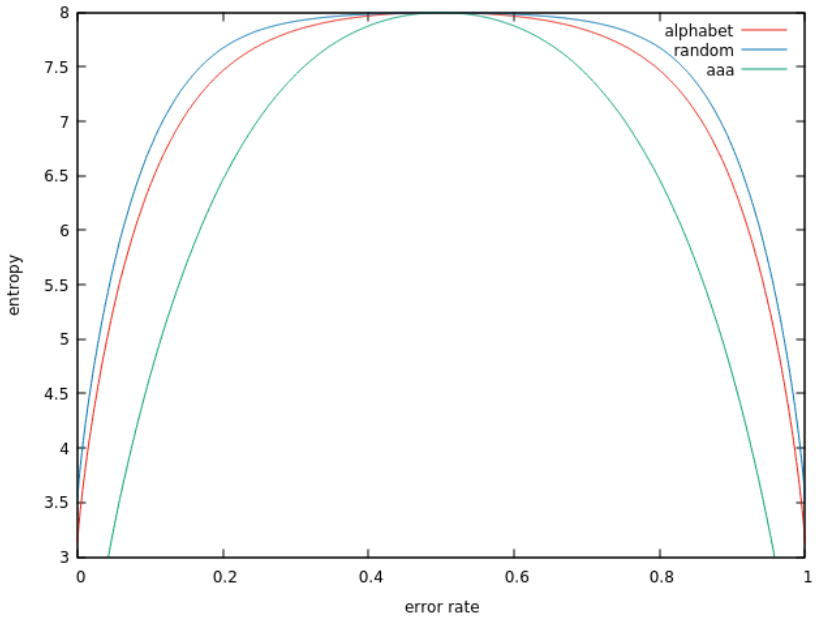

2. *alphabet*



3. *random*




- From all of the above graphs, it can be seen that entropy has a downward parabolic shape.
- The reason for this is that after the error rate goes above 50%, the error program starts flipping the bits (XORing) to their original values.
- Thus, the entropy/randomness of a file starts to decrease symmetrically.
- In fact, the higher the entropy of a given file at 0.0 error rate, the wider the parabola will be since the random bits are spread more apart. This could also be seen in the following graph:

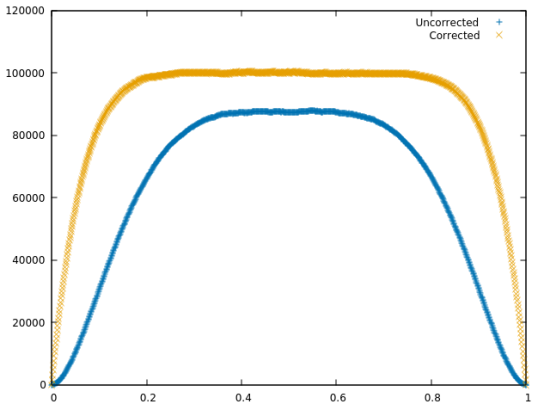


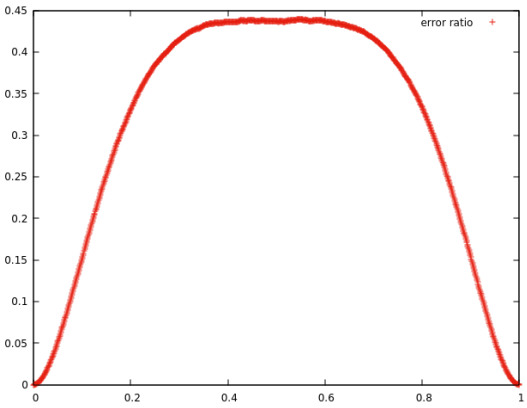
**Error Rate v. Decode Statistics**

- After the above experiment, I wanted to see if the decode stats such as uncorrected and corrected bits had similar graphs.
- The files are the same as described in the previous section.
- My initial guess was that regardless of the entropy of a file before being encoded, all the graphs should have the same number of uncorrected and corrected bits.
- The reasoning behind it is that the encoder does not really take entropy into account and since the entropy decreases symmetrically as the error rate increases after 0.5, any file of the same size should render the same statistics.

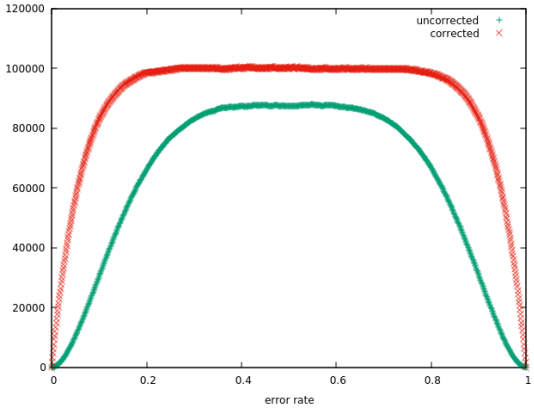
Graphs: (The graph on the left has the uncorrected and corrected stats; the right one has the error ratio (uncorrected/bytes processed).

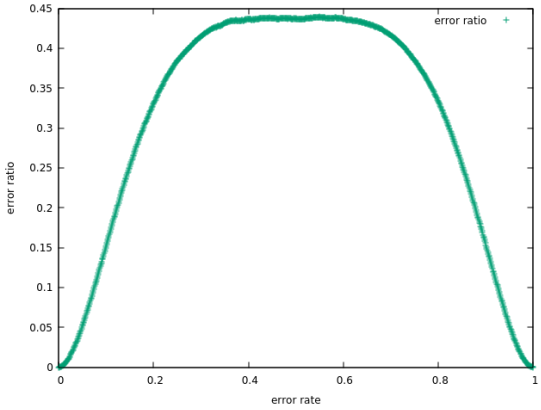
1. *aaa* file



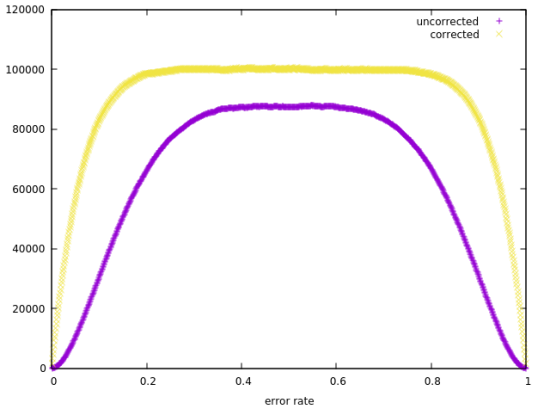



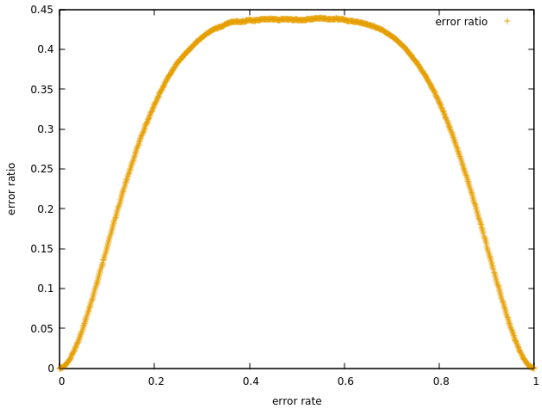

2. *alphabet* file



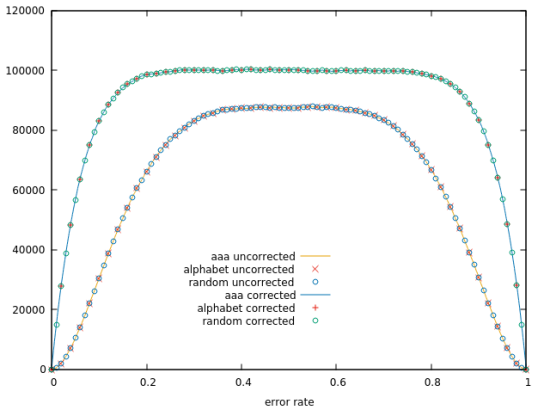
3. *random* file

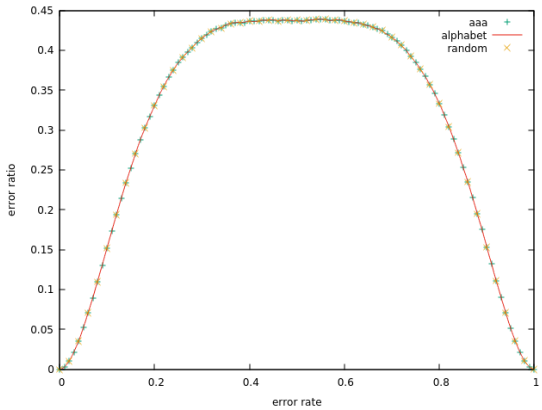



- From the above graphs, as assumed before, it can be seen that all of the graphs follow the same pattern with the same x and y boundaries.
- To get a more accurate picture, I graphed all the statistics (and error ratios) together.

4. All files




- Thus, as guessed and explained before, the assumption that each file of the same size will manifest the same statistics holds true as shown in the graphs above.

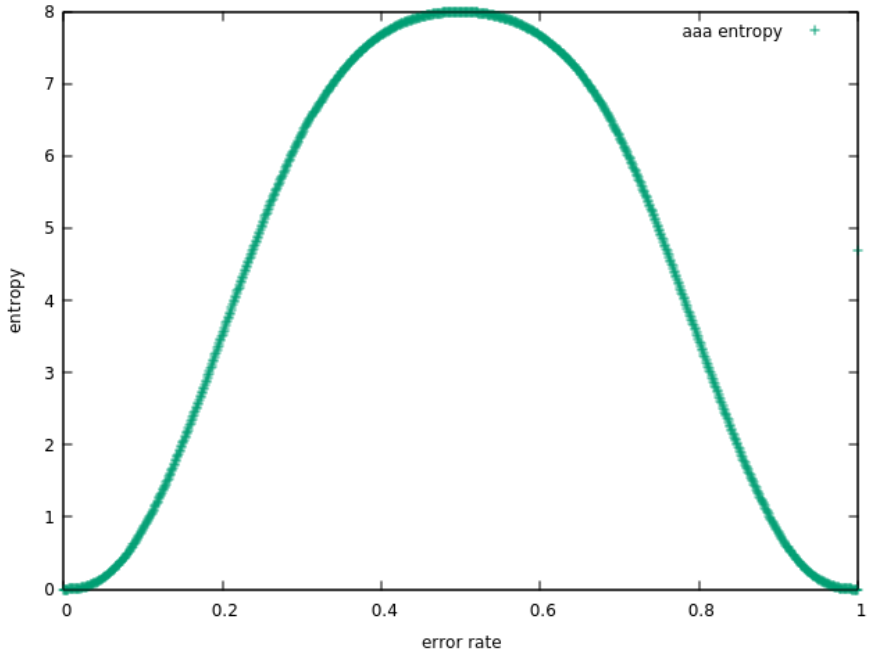
**Error Rate v. Entropy (after encoding and decoding)**

- Following the second section *Error Rate v. Entropy (before encoding or decoding)*, I plotted the graphs for entropy after a file has been encoded.
- I wanted to see whether the graphs had the same parabolic distribution as the file before encoding it.
- I was suspicious of this idea because of the entropy behavior of a file before and after encoding it as described in the first section *File Size v. Entropy*.

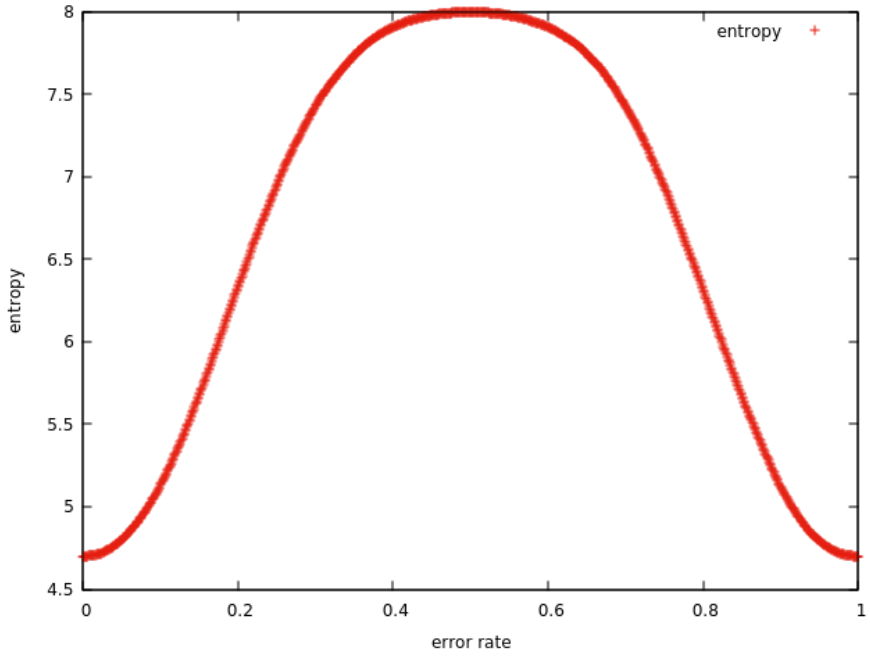
- In this case, I had a speculative initial guess. I guessed that the graphs should look similar to the second section since the file is being encoded and decoded and the only variable factor is the error rate.

Graphs:

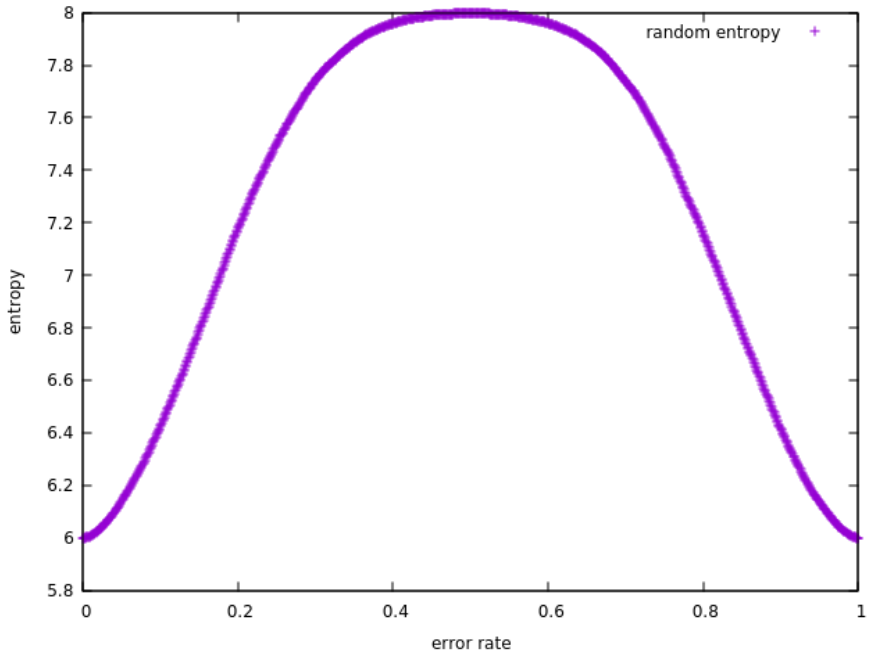
1. *aaa* file




2. *alphabet* file



3. *random* file




- Surprisingly, the graphs yielded totally different results. The graphs had a bell-ish curve shape.
- I realized what was wrong with the assumption before. The key idea here is that as the error rate increases not all the bits would be decoded. The parabolas would thus be stretched horizontally as the error rate increases.
- The reasoning behind it is that as the error rate increases not all of the bits would be corrected and some would be left as they were after encoding the file. Thus, the entropy or the randomness of the file would increase as the error rate increases (after a certain value) which explains the bell-ish graph.
- How much the graph would be stretched completely depends on the entropy of the file after it is encoded. Generally, the higher the entropy before, the more stretched the graph (and the higher it starts because of the initial value). This phenomenon can be seen in the graph below which marks the end of the writeup.

4. All files