Fabrice Kurmann
fkurmann@ucsc.edu
CSE 13s Spring 2021

Writeup for Assignment 5, Hamming Encoding/Decoding

In this assignment, I learned how to encode and decode files with a Hamming (8,4) encoder/decoder. Using this method with 4 parity bits, it is possible to detect and correct a majority of errors introduced in between the encoding and decoding processes. A smaller proportion of errors, those were more than one parity bit is positive can be detected but not corrected automatically.

After building the encoder and decoder, I used the Error and Entropy programs provided by Professor Long to introduce random error into encoded files and measure the entropy of my output files, respectively.

The error file works by replacing various bits of the file it is fed with random or swapped values. The bits that are corrupted are determined by a random generator with a specific seed.

The entropy test results in a numerical value representing the variance and variety of the file it is fed. A higher level of entropy leads to a higher numerical output while a lower entropy leads to a low output. For example the string “AAAA” has a lower entropy than “AABB” which in turn has a lower entropy than “ABCD”.

Summarized below are my observations after having played around with different files to encode and decode, different error rates and seeds, and measuring their effects on both the Hamming error rates (corrected vs uncorrected) and the entropy of both encoded and decoded outputs.

My Test Files:
I created three test files with differing levels of entropy, one designed to be low, one medium, and one very high. These are the files I used to create all the graphs below in this write up.

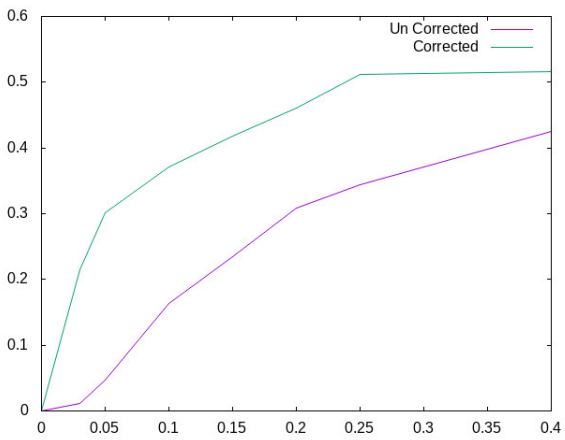
Below is a table showing the entropy levels for these files in their encoded and natural states.

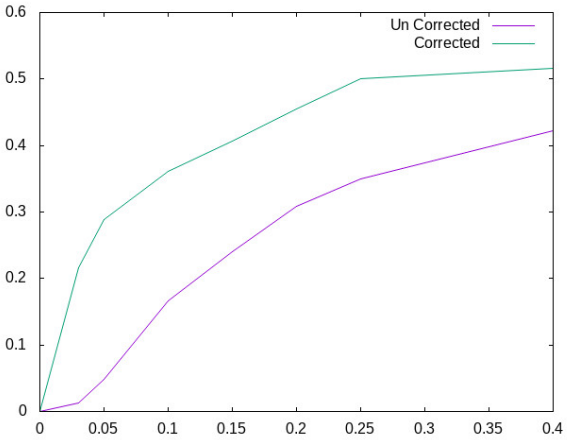
Test Files	Low Entropy File	Medium Entropy File	High Entropy File
Natural Entropy	0.1154	4.1678	6.5120
Encoded Entropy	1.0985	3.0750	3.7324

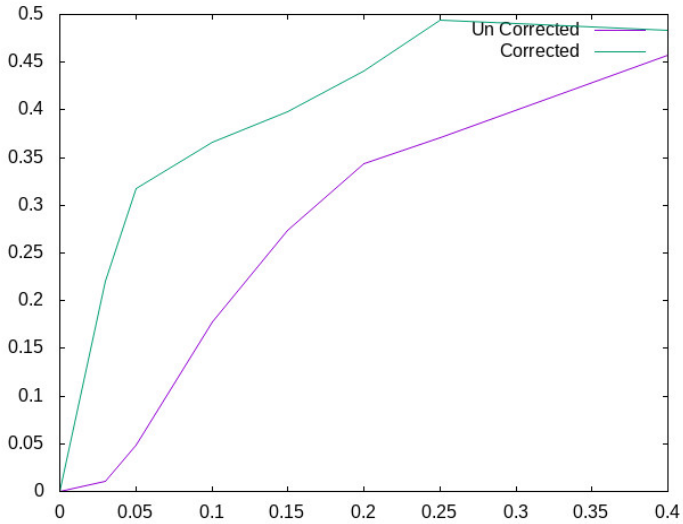
Hamming Decoding Given Introduced Error

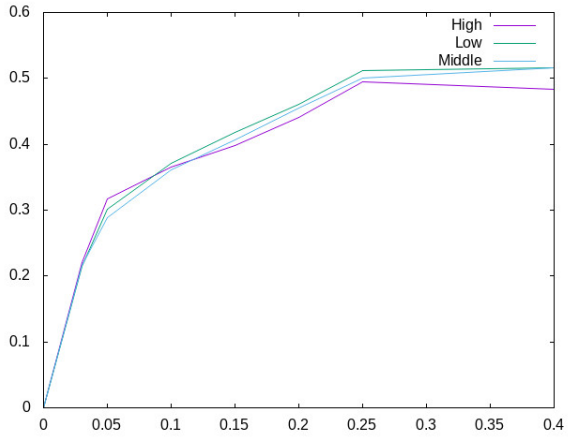
The first question I sought to answer to understand the power of the Hamming (8,4) method was: Of the errors the decoder caught, what proportion of those errors could be corrected and what proportion were truly lost.

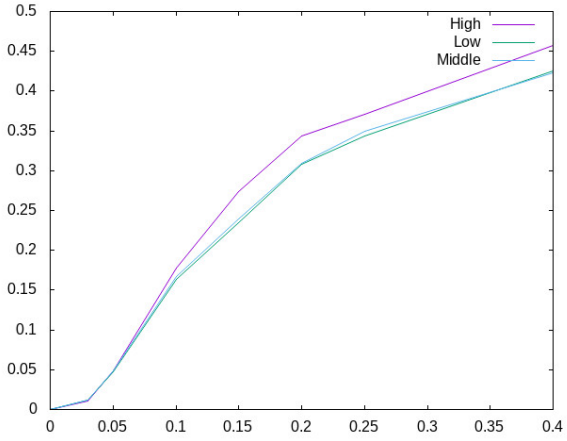
To see if there was a pattern I decided to graph the number of corrected and uncorrected errors based on an independent variable of input to the error function. Doing this across my three different entropy level function yielded the following graphs:











Above from top to bottom, left to right: Corrected and uncorrected error rates given differing error function inputs for low, medium and high entropy files.
Last two pictures show the uncorrected and corrected error rates separately but with all three files in one graph.

Some initial observations: the corrected error rate is consistently higher than the uncorrected, however, it seems to be the case that especially at lower error rates, uncorrected errors grow very slowly, beginning to catch up at around a 5% error insertion rate. Secondly, the uncorrected error rates seem to level off from when 25% to 40% error is introduced. I will discuss this further in the following section on error curves. Meanwhile the corrected error rates continue to rise, so as to almost meet the uncorrected rate at 40% introduced error. Finally, using the lower two graphs it is fairly clear that initial entropy does not affect the way error insertion results in errors after decoding.

As for my initial question of what the correlation is between insertion error and decoding error; these graphs look like logarithmic graphs. Both rise at a faster rate initially (excluding the slower start for uncorrected error) and slow down at introduced error of 20%.

Does this make sense? Well, given that the error program conducts bitwise flips on the data it is fed depending on the results of the random generator it makes sense for error the rate of corrected and uncorrected errors to rise as introduced error is increased. What is interesting however is that total errors (corrected and uncorrected) occur at about twice the rate as the input to the error function.

When considering what causes an correctable and uncorrectable error however, this becomes easy to explain. In order to cause a correctable error, only one of the 8 encoded bits must be flipped. Similarly, to cause an uncorrectable error two or more bits must be flipped. Thus, the rate of total errors may very well be twice as high as the input to the error function.

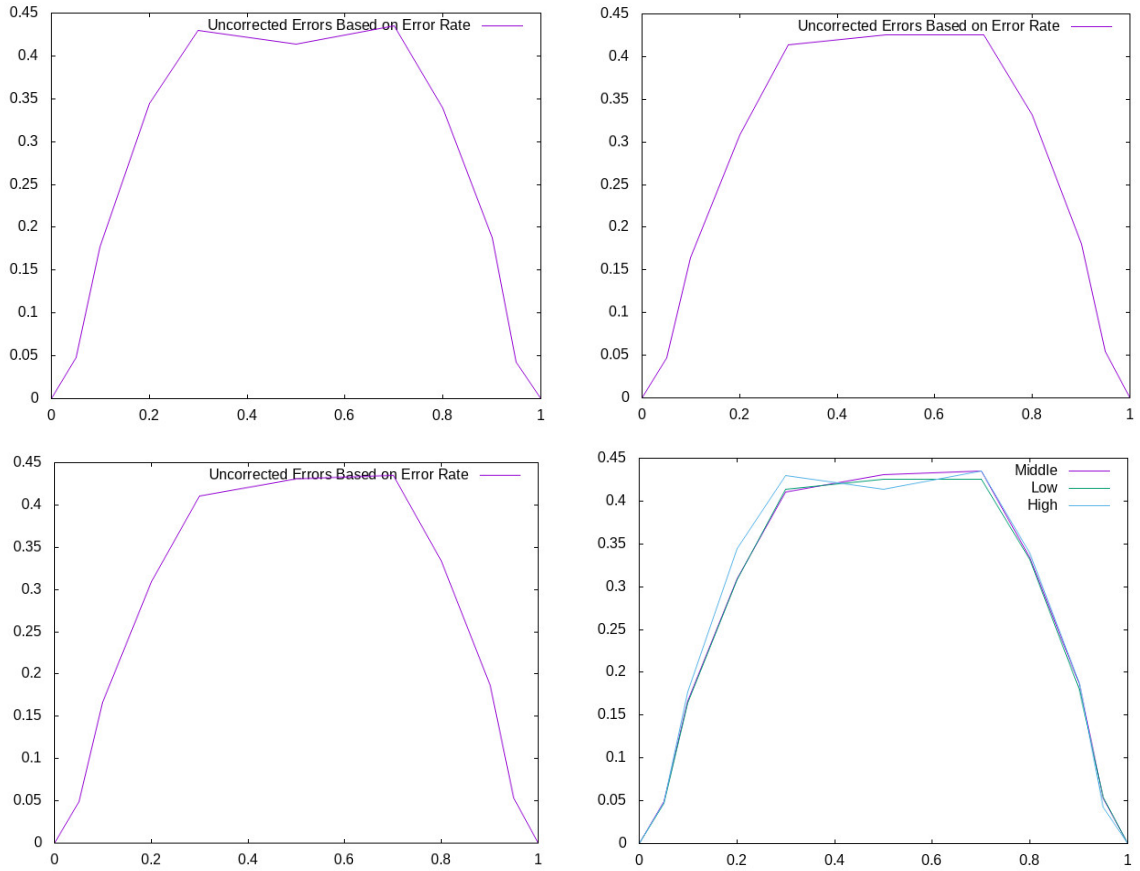
Hamming Decoding Given Introduced Error (Curves)

Next, how does error rate, as specified into the error function translate into error in the decoded files when I take error all the way to 100%? To test this I decided to graph the uncorrected error rates after applying error at values of 0, 0.05, 0.1, 0.2, 0.3, 0.5, 0.7, 0.8, 0.9, 0.95, and 1. My hypothesis was that as error rate increased , so would the amount of errors in my files after encoding and decoding.

However, after graphing the results I was surprised to find a bell shaped curve similar to that of entropy that I will discuss below. This seemed to me like an error, especially considering that the output files when error was high were unrecognizable compared to the input file. Why then was the error rate going down? Clearly there was a mistake happening somewhere. However, the difference with this mistake was that it was not only happening to my code. When running the same test on the code in the resources repo, I got the same contradictory results.

What causes uncorrected error rates to decrease despite the encoding encountering errors? Once again, when considering what the error function does, bitwise flips, there may be an explanation. If error rate is 100%, all bits are flipped, the parity and the message bits. These completely flipped parity bits and message bits, when multiplied with the transpose matrix will

yield a no error output. However, the message bits will still have been flipped, meaning a faulty message is printed. Hence the bell shaped curves.



Above top to bottom, left to right: High, medium and low entropy files' response to error input at various rates. Last image shows the comparison of all three on one graph.

Some further observations about these images: As discussed in the previous section, uncorrected error does not seem to rise over a threshold of 45%, leveling off and in the case of the high entropy file even dipping between 25 and 75% introduced error.

Effects of Encoding/Decoding on Entropy

Next, I wanted to determine if there was a general correlation between the entropy of files before and after they were encoded. My hypothesis here was that there would be a general trend of higher entropy in encode files. That said, since the encoder could be fed a file with entropy ranging from as low as possible (0.1154) to extremely high (6.5120).

I assume there are certainly input files, especially those with high levels of entropy, which could become more homogeneous through encoding. On the low end, given that the encoder simply divides each input byte and attaches the corresponding hamming parity bits to it, it is clear that

for very homogeneous input files, entropy will grow simply through the splitting of the input byte into two different output bytes.

This is where the table of entropy results for my test files becomes more interesting.

Test Files	Low Entropy File	Medium Entropy File	High Entropy File
Natural Entropy	0.1154	4.1678	6.5120
Encoded Entropy	1.0985	3.0750	3.7324

More Test Files	Random File 1	Random File 2	Random File 3	Random File 4
Natural Entropy	4.9743	4.8382	4.3769	4.8335
Encoded Entropy	3.4499	3.3292	3.3434	3.2285

As is apparent here, the trend is as I predicted: For files with high entropy, encoding shrinks entropy, while for those with exceptionally low entropy, encoding raises it. More middle or the road files will go either way, in this case entropy decreasing slightly.

What is interesting is that when encoding functions, the entropy seems to approach a limit around 2, at least based on these three results. Could it be that three or thereabout is a natural amount of entropy? To examine this further I generated a few more random files, where entropy was usually around the middle range and encoded them to see if a trend could become clear.

Based on this table next to the previous one, it appears my prediction about entropy values converging towards a limit of three was accurate if somewhat incomplete. *Encoded* entropy values do approach a limit near 3, in the case of these random files more specifically around 3.33. However, random files in their *natural* states all had entropy values in the 4.8 region.

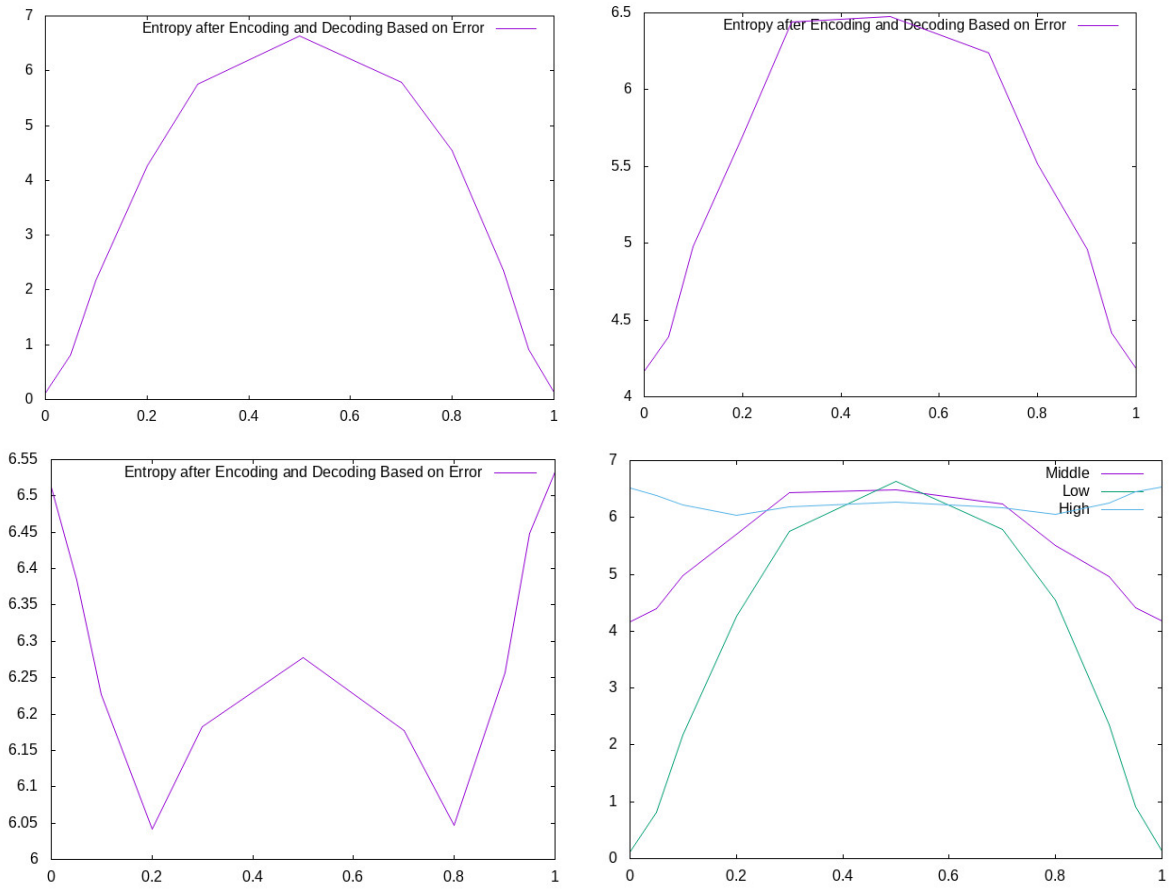
Why is the entropy of these natural random files higher than their encoded counterparts? This could have to do with the fact that to create the random files, I minimized repeated keys or patterns, something which in a truly random scenario could happen more often.

Effects of Differing Error Rates on Entropy

Guided by Eugene's lab section, I wanted to delve deeper on the fascinating trend demonstrated where the entropy of decoded files grew and then shrank again in comparison to the original file's entropy value as the error rate increased. Not only was the entropy based on error rate somewhat symmetrical, but the entropy of a file with 100% errors nearly matched the entropy of the original file.

To examine this, I decided to begin by plotting an entropy curve for a random file, encoded and decoded again after inserting differing amounts of error. I assumed this curve would be bell shaped with greatest entropy at 50% error based on the lab section demonstration.

After applying this encode - insert error - decode loop for error values in the range 0, 0.05, 0.1, 0.2, 0.3, 0.5, 0.7, 0.8, 0.9, 0.95, and 1 across my three different test files of different natural entropy I got the following curves.



Above top to bottom, left to right, entropy rates based on introduced error rate for low, medium, and high entropy files, followed by the combined graph.

These graphs are by far those with the greatest differences between the three files, with the high entropy file looking more like a “W” than a bell curve. However, when looking at the combined graph and the middle of the “W” shaped graph, it becomes clear that this different shape is only caused by the initially much higher entropy of this file. As soon as an error rate is applied, the graphs converge and once again look very similar in the 30 to 70% introduced error range.

The reason for the lower entropy after 50% error is the same as why uncorrected error drops off after 50% input error explained above. If all bits of the message bytes are flipped, the parity and the message ones, then the entropy between bytes will match the entropy between bytes of the

data before the encoding/decoding cycle. This is like a large example of a double negative: if everything is negated in the same way, it will still have the same correlation with itself, the only difference will be that everything looks different from the outside.

Additionally a few interesting observations are:

1. It seems to be the case that the upper limit for entropy is around 6.5. Firstly, this was the entropy of the file I created to have a deliberately high entropy. Then, as is very clear in the combined Entropy Curve Graph, all starter files hit levels of entropy around 6.5 when at the top of their entropy curves, when they were being inserted with an error rate of 0.5.
2. This is especially interesting when taken in context of the fact that the limit for encoded entropy is around 3.33. 6.5 is close to 6.66, two times 3.33. Is this coincidence or does this give an insight into how entropy is being quantified by this particular provided function.

Conclusion:

This assignment was the first I've done that involved encoding and decoding. After reading into it and thinking it over, I quickly understood the way Hamming (8,4) encoding works and how the parity bits are able to detect and in many cases even correct error. This is really powerful and a great mathematical trick. Additionally, the use of a matrix to abstract the encoding initially seemed crazy, however, when playing around with it, it became clear that this is a remarkably helpful connection between linear algebra and CS.

When using the provided error and entropy functions to test the limits of my encoder and decoder, I was able to discover limits to this method, most notably what begins to happen when more than half of the encoded bits are flipped. Could a more complex program be able to detect this is happening and flip the bits back even in these higher error ranges? For the simple case of 100% error rate, this seems feasible, so perhaps it is possible.

The concept of entropy is also an interesting connection, allowing for greater understanding of the way bit flips will affect the larger bytes that are being dealt with.