Lecture 13: Error correcting codes

ENGR 76 lecture notes — May 14, 2024

Ayfer Özgür, Stanford University

1 Error correcting codes

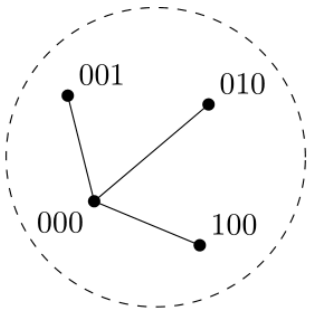
Recall that one motivation behind the use of digital communication techniques is to restrict the space of possible transmissions in order to get reliable communication. In the above scheme, by restricting the energy of each symbol to take one of two distinct values, we can tolerate fluctuations due to channel noise. For example, as long as the energy of the symbol stays below the threshold E_T when we transmit a "0", we will decode it as a "0". Similarly, as long as the energy of the symbol stays above the threshold E_T, it is decoded as "1". However, in some cases this might fail. For example, in the audio communication project, background noise like talking can lead to a "0" being decoded as "1". Similarly if the microphone gets obstructed for part of the communication, it might lead to some `1`s being decoded as `0`s. Error correction codes provide a further layer of protection by restricting the set of binary strings that can be transmitted.

The simplest error correction code is a **repetition code**. Assume that instead of transmitting each bit as it is, we repeat it three times. In the language of error correction codes, we assign the codeword `000` to the message "0" and the codeword `111` to the message "1". Observe that we use 3 bits to communicate a single bit. To understand why this is useful, consider what happens when one of the bits in the transmitted codeword gets flipped. We can still recover the transmitted codeword. Both that if we transmit `000`, and zero or one bits get flipped, we will still have more `0`s than `1`s. Similarly we transmit `111`, and zero or one bits get flipped, we will still have more `1`s than `0`s. Thus, we can perform a majority vote at the receiver to determine whether we have more `0`s or `1`s, decoding to 0 or 1 in the respective cases. Note that this won't be possible if we used only 2 bits.

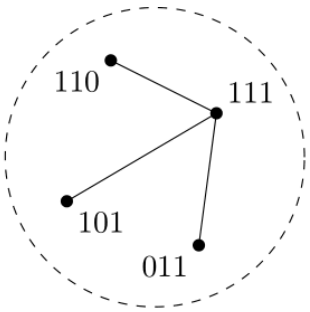
Digging further, we observe that

- If `000` is sent, we can receive `000`, `001`, `010` or `100`.
- If `111` is sent, we can receive `111`, `110`, `101` or `011`.

This can be shown graphically as shown below. In the figure, we draw *noise balls* around each codeword that consists of all the sequences that can be received when that codeword is sent and a single bit flip occurs . Since we have no intersection between the balls, we can communicate reliably. More formally, the *noise ball* is referred to as the **Hamming ball** of radius 1 around the codeword. The Hamming ball is similar to usual ball we draw in Euclidean spaces (e.g., the disk in 2D, the ball in 3D), but the distance measure is the **Hamming distance** and the Hamming ball of radius 1 around the codeword consists of all sequences that have Hamming distance of 1 or less with the codeword. Note that the Hamming distance between two binary sequences of the same length is defined as the number of places where they differ.



noise ball around 000



noise ball around 111

By increasing the number of times we repeat the bits, we can gain the ability to correct larger number of bit flips. Next, we will see some other codes that can outperform repetition codes.

Question 1: How many bits do we need to communicate one out of four possible messages, if one of the bits might get flipped?

Here we have the number of possible messages (M) equal to 4, or equivalently, we have 2 information bits. Our first attempt is based on the repetition coding idea we saw before, where we just represent the two information bits thrice.

Code I:
Message 1: `000000`
Message 2: `010101`
Message 3: `101010`
Message 4: `111111`

This code has codeword length of 6 bits, and can correct one bit error using majority vote decoding. Can we do better? Consider the code below:

Code II:
Message 1: `00000`
Message 2: `11100`
Message 3: `00111`
Message 4: `11011`

This code has codeword length of 5 bits, which is better than Code I. But how do we verify that this works reliably even if one bit is flipped? One way is to find the Hamming balls of radius 1 around all four codewords and verify that they do not intersect. Equivalently, we can simply check if the Hamming distance between any two codewords is at least 3, which guarantees that after one bit flip we are still closer to the true codeword as compared to any other codeword. Thus, we can recover the true codeword by looking for the codeword with the minimum Hamming distance to the received sequence. Let's look at the Hamming distances for the pairs of codewords (note that Hamming distance is symmetric so we only have 6 pairs):

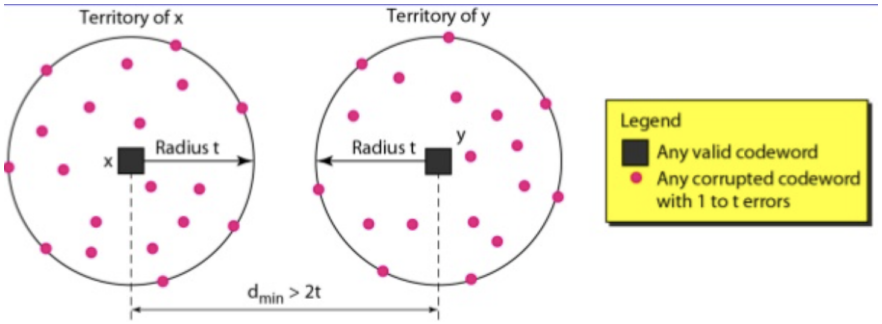
- `00000` and `11100`: Hamming distance 3
- `00000` and `00111`: Hamming distance 3
- `00000` and `11011`: Hamming distance 4
- `11100` and `00111`: Hamming distance 4
- `11100` and `11011`: Hamming distance 3
- `00111` and `11011`: Hamming distance 3

We define the **minimum distance** of a code as the minimum Hamming distance between any pair of (distinct) codewords. This is also referred to as the **distance of the code**. For code II, we checked that the (minimum) distance is 3, which shows that the code can correct 1 bit flip.

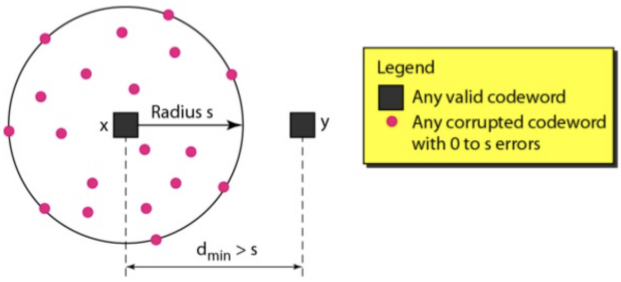
There are three figures of merit for an error correction code we saw last time:

- The number of strings is the size of the code (M). The example has $M = 4$. We want this to be big so we can communicate more information bits.
- The length of the strings is the length of the code (L). The example has $L = 5$. We want this to be small so we need to use the channel for a shorter duration.
- The minimum Hamming distance between the strings is called the distance of the code (d). The example has $d = 3$. We want the distance to be big because that allows recovery from a larger number of bit flip errors.

Let's try to understand the relation between the minimum distance of the code and it's error correction capabilities. In the last lecture, we saw that for correcting 1 bit flip, we need a distance of 3. More generally, to correct t bit flips, we need distance of at least $2t + 1$. This is illustrated in the figure below, basically what we need to make sure is that the noise balls (Hamming balls) around two codewords do not intersect which guarantees we can always recover the original codeword.



Another problem of interest is error detection, where we only care about being able to tell whether or not a bit flip occurred. In such cases, it is possible to show that we need a minimum distance of at least $s + 1$ to be able to detect up to s errors (illustrated in the figure below from the same source).



In general, we are interested in finding the best tradeoffs between M, L and d. For example, let's revisit the code above {`00000`, `00111`, `11011`, `11100`} with $M = 4$, $L = 5$ and $d = 3$. Is it possible to have a code with $L = 4$ for the same M and d? Let's say such a code exists. Then we can consider the Hamming ball of radius 1 around each codeword. The ball consists of 5 elements, the original codeword and the 4 Hamming neighbors obtained by flipping one bit in the codeword. Since the code has distance 3, it is able to correct 1 bit flip, and the Hamming balls should not intersect. Thus the 4 Hamming balls around the codewords, each with 5 sequences must be disjoint, giving us 20 unique sequences. This is a contradiction since we have only 16 sequences of length 4! Hence, it is impossible to achieve $M = 4$, $L = 4$ and $d = 3$. This sort of packing argument is quite general, and is called the Hamming bound. In general, finding the best tradeoff is an open problem, but we will see some specific examples of good codes in the next lecture.