MA294 Lecture

Timothy Kohl

Boston University

April 4, 2024

Error Correcting Codes

Suppose there is a space probe in orbit around Mars being controlled by a ground station on Earth.
Commands are being sent to this probe by radio which the probe responds to.

The problem is that solar flares or other such interference can affect the signal, potentially garbling the message.

How does one insure that the correct commands and information are being transmitted?

Furthermore, how can we insure that the probe can deal reliably with this interference, and the resulting potentially garbled messages?

Information like this is typically encoded in binary $\{0, 1\}$ or, more specifically, strings of '0' and '1' of some fixed length.

We can view these fixed length 'code words' (which has nothing to do with encryption) as vectors $v \in V = \mathbb{F}_2^n$ (the vector space V of dimension n over $\mathbb{F}_2$) for some n.

i.e. bit 'strings' of some fixed length.

Example: $n = 3$, $V = \{000, 001, 010, 011, 100, 101, 110, 111\}$

Generally, the codewords are usually some fixed subset $C \subseteq V$, say $C = \{000, 100, 001, 010\}$.

There are two considerations:

- the number of codewords that one needs for the communcations channel
- the code length to use for these codewords

and the point is that the second depends on the first, in particular if we wish to detect if an 'error' has been made, namely that a 0 is received instead of a 1 in a given codeword, and vice versa.

The initial idea is to choose the codewords of a given length so that an error is 'obvious'.

A crude method for transmitting a '0' or '1' is to use 111 for '1' and 000 for '0', so that if an error occurs, the correct interpretation can be made by 'majority rule', namely

$$\begin{aligned} \text{received} &\rightarrow \text{interpreted} \\ 000 &\rightarrow 0 \text{ i.e. no erorr} \\ 001 &\rightarrow 0 \\ 010 &\rightarrow 0 \\ 100 &\rightarrow 0 \\ 111 &\rightarrow 1 \text{ no error} \\ 110 &\rightarrow 1 \\ 101 &\rightarrow 1 \\ 011 &\rightarrow 1 \end{aligned}$$

So this code is ok if no more than one error occurs, but if say two bits are flipped in 000 to yield say 110 then the 'majority rule' interpretation will fail. Also, this code is extremely inefficient since one generally needs bit strings longer than one bit in length.

Let's compare a couple of different examples, each of which is set C of four codewords, which are subsets of different $\mathbb{F}_2^n$.

$C_1 = \{00, 10, 01, 11\}$

So here, then length of each codeword is 2, so that $C_1 \subseteq V = \mathbb{F}_2^2$.

But since $|V| = |\mathbb{F}_2^2| = 2^2 = 4$ then this means *every* bit string in V is a codeword.

This is not a good choice at all since any error will transform a given codeword into a different codeword, so a receiver will be unable to tell that the erroneous codeword *is* erroneous.

So we observe that this code cannot detect *any* errors.

$C_2 = \{000, 110, 011, 101\}$
Here $C_2 \subseteq V = \mathbb{F}_2^3$ so it's not all of V, and is a better choice because any single error, in any codeword, will result in a bitstring *not* in C_2.

For example

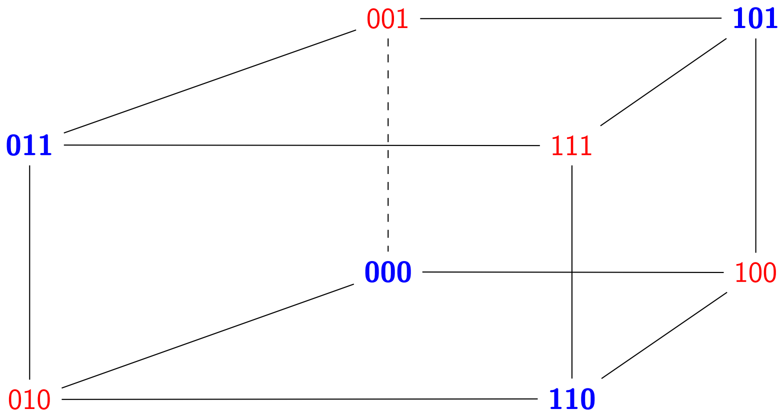
$$000 \to 100 \notin C_2$$
$$000 \to 010 \notin C_2$$
$$000 \to 001 \notin C_2$$

The one disadvantage of this code is that it cannot *correct* a given one bit error. Why?

$C_2 = \{000, 110, 011, 101\}$

Consider the error $000 \rightarrow 100$, which is not in C_2.

However, if one wanted to infer what the correct codeword that was meant to be transmitted, one cannot do this since if one 'corrects' the first bit $100 \rightarrow 000 \in C_2$, but if one corrects the second bit, they get $100 \rightarrow 110 \in C_2$ as well.



This ambiguity is a symptom of the fact that the codewords are in some sense (which we will make precise) too *close* to each other in order to correct an erroneous codeword back to *one and only one* correct codeword.

A still better set of codewords is this:

$$C_3 = \{000000, 111000, 001110, 110011\} \subseteq V = \mathbb{F}_2^6$$

Here, a single bit error is not only detectable in that it results in a bit vector not in C_3 but is also *correctable*.

Why?

The basic idea is that if $\vec{c}$ is a codeword in C_3 and we change one of its bits to get an 'erroneous' vector $\vec{b}$ and if we take any other codeword $\vec{c}'$ and change any of its bits to get $\vec{b}'$ then it is **never** the case that $\vec{b} = \vec{b}'$ and so there is basically no vector that is a single bit away from $\vec{c}$ except $\vec{c}$ itself!

i.e. There is a sense of 'distance' between codewords which we'll make precise and use to formalize all this intuition.

Definition

For $\vec{a}, \vec{b} \in V = \mathbb{F}_2^n$, the Hamming <u>distance</u> $\partial(\vec{a}, \vec{b})$ is the number of bits that differ between $\vec{a}$ and $\vec{b}$.

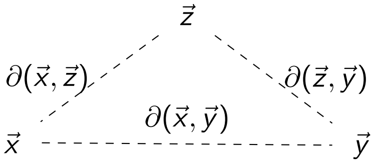
For example $\partial(1\mathbf{0}1, 1\mathbf{1}1) = 1$

Lemma

The distance function ∂ satisfies the following properties:
(i) $\partial(\vec{x}, \vec{y}) = 0$ if and only if $\vec{x} = \vec{y}$
(ii) $\partial(\vec{x}, \vec{y}) = \partial(\vec{y}, \vec{x})$
(iii) $\partial(\vec{x}, \vec{y}) \leq \partial(\vec{x}, \vec{z}) + \partial(\vec{z}, \vec{y})$, for $\vec{x}, \vec{y}, \vec{z} \in V$

The first two statements are relatively obvious, and are an exercise.

The third statement is referred to as the triangle inequality and is, as the name suggests, geometric in nature as we are viewing the vectors $\vec{x}, \vec{y}, \vec{z} \in V = \mathbb{F}_2^n$ as being in a space where the distance function ∂ follows rules like how distances behave in ordinary Euclidean space.


$\vec{z}$
$\partial(\vec{x},\vec{z})$
$\partial(\vec{z},\vec{y})$
$\partial(\vec{x},\vec{y})$
$\vec{x}$
$\vec{y}$

$$\partial(\vec{x},\vec{y}) \le \partial(\vec{x},\vec{z}) + \partial(\vec{z},\vec{y})$$

i.e. The shortest distance between two points is a straight line.

For a given code C there is an important number which characterizes it.

Definition

For a code C, the minimum distance between pairs of codewords is

$$\delta = min\{\partial(\vec{a},\vec{b}) \mid \vec{a},\vec{b} \in C \ ; \vec{a} \neq \vec{b}\}$$

- For $C_1 = \{00, 10, 01, 11\}$, $\delta = 1$
- For $C_2 = \{000, 110, 011, 101\}$, $\delta = 2$
- For $C_3 = \{000000, 111000, 001110, 110011\}$, $\delta = 3$

Observation: The distance between two codewords is the number of errors needed to turn one into the other.

So, for a given code C with minimum distance δ, if no more than $\delta - 1$ errors are made in transmission, the receive will be able to detect that an error has been made.

But what about *correcting* errors?

Correcting Errors

The scheme we will explore is the nearest neighbor decoding principle, namely that if an erroneous codeword was transmitted, then we can assume that the nearest codeword in C is the correct one.

Theorem

A code C will correct 'e' errors by the nearest neighbor principle provided that the minimum distance δ satisfies the inequality

$$\delta \geq 2e + 1$$

.

The proof of this isn't too difficult, and makes nice use of the triangle inequality we saw earlier.

Proof.

Say $\vec{c}$ was sent and e errors were made, yielding the bit vector $\vec{z}$.

As such we have $\partial(\vec{c},\vec{z}) = e$.

If now $\vec{c}^{'}$ is any other valid codeword then

$$\partial(\vec{c},\vec{z}) + \partial(\vec{z},\vec{c}^{'}) \geq \partial(\vec{c},\vec{c}^{'}) \geq \delta \geq 2e+1$$

Thus $e + \partial(\vec{z},\vec{c}^{'}) \geq 2e+1$ and so $\partial(\vec{z},\vec{c}^{'}) \geq e+1$ which implies that $\vec{c}$ is the nearest codeword to $\vec{z}$ and so by the nearest neighbor principle, we infer that the correct codeword *is* $\vec{c}$. □

Going back to the example:

$$C_3 = \{000000, 111000, 001110, 110011\}$$

since $\delta = 3$ then it can detect 2 (i.e. $\delta - 1$) errors and correct $e = 1$ errors, since $\delta = 2(1) + 1$.