Linear and Cyclic Codes

Hassam

The Basic Problem (revisited)

- Sending messages between parties might be subject to errors.
- How do we:
  - Detect Errors?
  - Correct Errors?
- The goal is to send a message that minimizes the necessary redundancy to achieve both goals.
- What if we only care about certain common types of errors?

Section 1

Mathing up Hamming Codes

Reminder of Hamming Codes

- We can detect the error by checking each parity bit, and fix it!

0	1	1	0
1	1	0	0
0	1	0	0
1	0	1	1

0	1	1	0
1	1	0	0
0	1	0	0
1	0	1	1

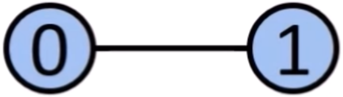
0	1	1	0
1	1	0	0
0	1	0	0
1	0	1	1

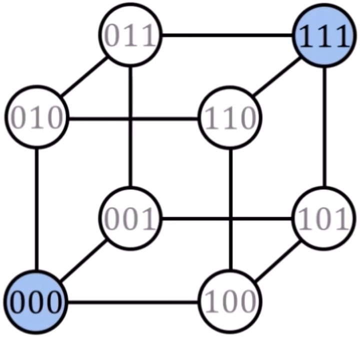
0	1	1	0
1	1	0	0
0	1	0	0
1	0	1	1

Going Higher

- What does it really mean to correct an error?
- Let's consider a simple code: "best 2 out of 3".
- Encode message m by taking each bit and repeating it three times.
- Our valid "words": 000 and 111, but the language contains 2^3 words.
- We are raising the dimension of our message space, giving us more room to detect errors.

Going Higher





Hamming (7, 4) (because the dimensions are smaller)

- (n, k) *code* means n bits to encode a k bit message.
- Consider a message with 4 bits. We can use 3 additional bits of information to "search" over these bits and detect errors, just like the earlier Hamming example.
- $p_0 = m_0 + m_1 + m_3$, $p_1 = m_0 + m_2 + m_3$, $p_2 = m_1 + m_2 + m_3$. All mod 2.
- Convince yourself that we can detect, and find, any 1-bit error using these check bits.

Linear Equations $\implies$ Linear Algebra

Can we create a matrix G that converts our message into a code? $mG = c$

$$mG = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 & m_0 + m_1 + m_3 & m_0 + m_2 + m_3 & m_1 + m_2 + m_3 \end{bmatrix} =$$

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 & p_0 & p_1 & p_2 \end{bmatrix} = c$$

Linear Algebra $\implies$?

- Turning Hamming Codes into a matrix gives us linearity for free!
- Namely, the sum of any two messages is a valid message, and, by linearity, the sum of any two codewords is also a codeword.
- Let's see if we can prove some nice properties using this.

Minimum Distance (revisited)

- The *Minimum Distance* of a code is the minimum difference between any two codewords.
  - $\Delta(x, y)$ is the number of positions i where characters $x[i] \neq y[i]$
- More formally $\min\limits_{c_1, c_2} \Delta(c_1, c_2)$
- But this is just $\min\limits_{c_1, c_2} \Delta(0, c_2 - c_1)$
- But again, this is just $\min\limits_{c} \Delta(0, c) = \min\limits_{c} w(c)$

Parity Checking (but with linear algebra this time)

- Can we find a matrix, H, such that $Hc^T = 0$. Why?
- Linearity!
- Suppose we transmitted some code c with error e, $c + e$, then: $H(c+e)^T = Hc^T + He^T = He^T$. What is He^T?
- It is the column in H associated with the error
- Since we know where in H the error is, we know which bit of the message it corresponds to. We can both detect errors and correct them! We call this the "syndrome" vector.
- $HG^T = 0$ and so it follows that $H = \left[P|I_{n-k}\right]$. Verify this yourself.

Generalizing Hamming Codes

- Hamming codes are a code such that the Parity-Check matrix has columns with every possible combination of 0s and 1s, except for all 0s.
- So to make a Hamming code for higher dimensions, create a matrix of every single binary number, rotate them so that the identity is in the last columns, and the remaining columns are our parity checks.

Generalizing Hamming Codes

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Generalizing Hamming Codes

Moving the previous columns around to form a parity check matrix then gives into:

$$\left[\begin{array}{ccccccccccc|cccc}
0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1
\end{array}\right]$$

Each row covers one bit of the message, together covering them all.

Section 2

Cyclic Codes

Motivation

- More math!
- Most errors occur in “bursts”. You might have 5 errors in a message, but they will generally all be concentrated near each other.
- Working under this constraint, rather than detecting arbitrary errors, can give us a lot more power.

Cyclic

- *Cyclic codes* are relatively self explanatory.
- C is cyclic if $(c_1, c_2, c_3, \dots, c_n) \in C \implies (c_n, c_1, c_2, \dots, c_{n-1}) \in C$.
- Cyclic codes by themselves are not really useful

(Linear) Cyclic

- A much more useful and practical type of codespace is one where every codeword is cyclic AND linear.
- Example codespace: 000000, 100100, 010010, 001001, 110110, 011011, 101101, 111111.
- Every codeword can be written as the sum of two other codewords, and every sum of codewords is a codeword. Similarly, shifting any codeword will produce a valid codeword.

Polynomials (an aside)

- We can think of these codewords as a sequence, and create a polynomial called a *generating function*. For example: $100100 = 1x^0 + 0x^1 + 0x^2 + 1x^3 + 0x^4 + 0x^5 = 1 + x^3$.
- Polynomials have a lot of nice properties similar to the integers.
- Namely, we can do polynomial division, and modular arithmetic over polynomials. We will often write $\mathbb{Z}_2[x]$ to talk about polynomials with $\{0, 1\}$ as the coefficients.

Modular Arithmetic with Polynomials

- Consider $(x^3 + x^2 + 1)/(x^2 + 1) = (x^2 + 1)q(x) + r(x)$.
    - What are $q(x)$ and $r(x)$ if our polynomial is in $\mathbb{Z}_2[x]$?
    - $q = x + 1$, $r = -x = x$.
- So, $x^3 + x^2 + 1 \equiv x \mod (x^2 + 1)$.
- We will primarily work mod polynomials of the form $f(x) \pm 1$
    - So $f(x) \pm 1$ acts like 0
    - You can think of substituting every occurrence of $f(x)$ with 1.

Cyclic Codes fall out naturally

- Let's go back to one of our initial codewords: $100100 \equiv 1 + x^3$.
- The maximum degree of a polynomial in this codespace is x^5
- If we take our codewords mod $x^6 + 1$, then $x \cdot (1 + x^3) = x + x^4 \equiv 010010$, which is another codeword.
- We can continue to shift our codewords by multiplying by x, and if we go over x^5, our modulo wraps it back around! So, $c(x) \in C \implies xc(x) \in C$.

Linearity strikes back

Recall that our codewords are not just cyclic, but also linear. So, if $c(x) \in C$, then $c(x) + c(x) \in C$.

Since $x^n c(x) \in C$, this means that $(1 + x + x^2 + \ldots)c(x) \in C$. In otherwords, for any polynomial, $g(x)$, then $g(x)c(x) \in C$.

Generator Polynomial

If we can make other valid codewords using any polynomial, we just need 1 codeword to "generate" the rest. (I apologize for the differing uses of generate, blame whoever came up with this stuff).

Consider again: 000000, 100100, 010010, 001001, 110110, 011011, 101101, 111111. Can we generate all of the other codewords from 100100, or $g(x) = 1 + x^3$, mod $x^6 + 1$?

Generator Polynomial

$$\begin{aligned} 0 &= 0 \cdot g(x) \\ 1 + x^3 &= 1 \cdot g(x) \\ x + x^4 &= xg(x) \\ x^2 + x^5 &= x^2 g(x) \end{aligned}$$

$$\begin{aligned} 1 + x + x^3 + x^4 &= (1 + x)g(x) \\ x + x^2 + x^4 + x^5 &= (x + x^2)g(x) \\ 1 + x^2 + x^3 + x^5 &= (1 + x^2)g(x) \end{aligned}$$

and $1 + x + x^2 + x^3 + x^4 + x^5 = (1 + x + x^2)g(x)$. Is the generator polynomial within a codespace unique?

Generator Polynomial

- No it is not!
- We could have used any of the first three codewords and shifted things accordingly, but in general, we call the polynomial with the lowest degree the generator polynomial.
- We can encode a message of size k using a generator polynomial with max degree $r = n - k$, $c(x) = m(x)g(x)$.
- Since these are linear codes, we can create a generator matrix just like we did for linear codes.
- These are not super interesting, so I'll leave it up to you all to think about how to form them.

Correcting

- Briefly, if $c(x) = m(x)g(x)$, we can do $m(x) = c(x)/g(x)$ to get back our original message, if there were no errors.
- If there was an error, we'll have a remainder. For single-bit errors, just like our linear codes, we can create a table of remainders and match our remainder to figure out where the error occurred.
- But, this is no better than Hamming Codes.

Cyclic Redundancy Checks

We said earlier that cyclic codes can be used to detect burst errors. How? Let's consider this simple scheme, commonly used in networking, known as the *Cyclic Redundancy Check*.

- Take our message $m(x)$, and multiply it by x^r, where r is the degree of our generator polynomial for our codespace, and then divide it by our generator $g(x)$.
- We will end up with $x^r m(x) = Q(x)g(x) + R(x)$.
- Under $\mathbb{Z}_2$, addition and subtraction are the same operation $\implies Q(x)g(x) = x^r m(x) - R(x) = x^r m(x) + R(x)$.
- If we transmit $T(x) = x^r m(x) + R(x)$, this will be divisible by $g(x)$.

Cyclic Redundancy Checks $\overset{?}{=}$ Parity Bits

- We will only miss errors if they are divisible by $g(x)$, so we could select a $g(x)$ such that it is not divisible by "common" things.
  - Indeed, this is how most modern tooling using CRC does it. But let's not rely on that.
- Suppose we transmit $T'(x)$, where T' has some error that causes an odd number of flipped bits.
- Assuming that our initial message has an even number of 1s, if we've flipped an odd number of bits, then $T'(1) = 1$.
- So, as long as $g(1) = 0$, we will detect any odd number of bit flips. Picking $g(x) = x^i + 1$ will always ensure that $g(1) = 0$.

In fact, $g(x) = x + 1$ is equivalent to a parity check bit.

Cyclic Redundancy Checks

- So, we can detect any number of odd bit flips, just like a parity bit. What else can we do?
- Suppose we had a sequence of errors in a row.
- Let $E(x) = T(x) - T'(x)$ be which bits are flipped.
- $g(x)$ divides $T(x)$
  - $E(x)$ doesn't divide $T(x) \implies g(x)$ doesn't divide $T'(x)$.

What does $E(x)$ look like if we had a sequence of r errors?

$E(x) = x^{n_1} + x^{n_2} + \ldots + x^{n_r}$, where n_i is a decreasing sequence.

Math Magic

- So, we can factor $E(x) = x^{n_r}(x^{n_1 - n_r} + x^{n_2 - n_4} + \ldots + 1)$.
- $g(x) = x^i + 1$ cannot divide x^{n_r}. So, CRC can only fail if $g(x)$ divides $x^{n_1 - n_r} + x^{n_2 - n_r} + \ldots + 1$. How can we ensure that never happens?
- Choose $g(x)$ such that it's degree is larger than the number of errors you'd like to detect.

Is it worth the effort?

- Dividing polynomials is hard. I don't want to do it.
- The computer is more than happy to do it for you. Dividing polynomials over $\mathbb{Z}_2$ is just a problem of shifting bits and XORing them.
- CRC is fast at detecting and correcting errors of size 1, but it's true power shines in detecting any size of burst errors.

CRC is being used by every single device on the planet when it communicates over the internet. CRC is used the data layer and the transport layer of all computer networks. Sometime soon, we'll discuss more advanced codes (Reed-Solomon) that are used for a lot of other things you're familiar with, like barcodes and QR codes.

The field of 'information theory' began by using the old hardware paradigm of transportation of data from point to point.

— MARSHALL MCLUHAN (1988)