Lecture 11: Error Detecting and Correcting Codes

built on 2021/02/08 at 11:42:58

Today we're going to take a short break from the main trunk—instead, we'll look at a very practical question:

How to reliably transmit bits over a channel that can damage your bits?

Such a channel is known in the literature as a *noisy channel* and it may flip or lose some bits.

The general framework is depicted below:

```
Encoding:        Channel         Decoding:
             -----------------
m --> C(m)      => => => =>      y = C(m) + Error --> m
             -----------------
```

In the simplest form, a *message* m consists of k bits: $m_1m_2\dots m_k$. For this lecture, we'll discuss block codes, keeping the block size k fixed. The encoder converts the message into a codeword: $m \to C(m)$. The codeword travels through a possibly noisy channel, resurfacing on the other end as $C(m)$ + Error, i.e., potentially with some error. The decoder then attempts to restore the original message.

An encoding/decoding scheme can tolerate certain error conditions. We'll look at a few schemes and see what they're capable of handling.

Error detecting and correcting codes are used in a wide variety of applications. Here are some examples: Memory (ECC), Storage (CDs, DVDs, RAID), Wireless (cell phone and wireless links), etc.

Remarks: Not covered here, but... Reed-Solomon traditionally was most often used in practice, but recently (past 10-15 years) LDPC codes are becoming more widely used. Algorithms for decoding them are quite sophisticated.

1 Simple Parity Bit

Let's look at a really simple scheme first. To send a message m, the parity scheme involves appending a single parity bit to the message—that is,

$$C(m = m_1m_2\dots m_k) = m_1m_2\dots m_kp, \text{ where } p = m_1 \oplus m_2 \oplus \dots m_k.$$

It is this $C(m)$ that is sent over the channel. On the decoding end, we expect the received codeword y to have an even parity because $p = m_1 \oplus \cdots \oplus m_k$ and so $m_1 \oplus \cdots \oplus m_k \oplus p = 0$.

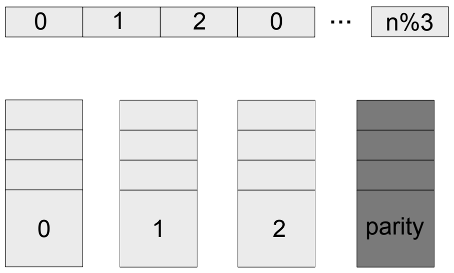
What can it do? It can detect any 1-bit error (i.e., a single bit flip anywhere), but it cannot correct the problem. Errors in more than one location? You're out of luck. Keep in mind that even if the party is correct, it doesn't mean the message is correct—there could have been an even number of flips.

Technically, it's capable of correcting some very limited form of error. In the literature, a bit lost is known as an *erasure*. If only 1 bit is gone and we know which bit is gone, say the i-th bit, then we can recover that $m_i = p \oplus m_1 \oplus \cdots \oplus m_k$.

1.1 Another Useful Application: RAID

Quite remarkably, one place where this shows up is in an idea called RAID: Redundant Array of Inexpensive Disks.

To improve disk read performance, one can split data over several disks. The data is grouped into stripes, which are spread across multiple disks. Below, we show a configuration with $3 + 1$ disks (3 data disks + 1 parity disk). With the parity bits stored, one can reconstruct the data if one of the disks were to fail (see erasure idea above).



This is nice and all, except that such a configuration creates a bottleneck in terms of writes: while reads can take place in parallel, each write request—no matter how small—has to involve the parity disk. In reality, there are schemes that spread parity bits rather evenly on all the disks.

2 Error Correction

Error detection is clearly useful, but better yet, we'd like to correct the error when detected. Otherwise, the data needs to be retransmitted anew, provided that the original still exists.

Say the channel can corrupt at most 1 bit every $k = 3$ bits. What can we do?

- *Repeating each bit twice* is enough to let us detect a bit was flipped but insufficient to correct the error.
- *Repeating each bit 3x* is sufficient to detect and correct one-bit flip. We can return the majority vote of the encoded string.
- But to send x bits of information, we end up sending $3x$ bits—quite wasteful!

Indeed, better schemes exist!

3 Hamming Codes

Richard Hamming worked at Bell Labs in the 1940s. He grew increasingly frustrated with having to restart his programs from scratch due to errors, which could be detected but not corrected. In an interview, he said "And so I said, 'Damn it, if the machine can detect an error, why can't it locate the position of the error and correct it?"' And over the next couple of years, he worked on the problem of error-correcting codes, leading to a publication in 1950 of what is now known as Hamming Code. Today, this is often used in ECC memory.

Let's introduce this code by way of example. To send a $k = 11$-bit message $b_1b_2\ldots b_{11}$, we're going to the send the following 15 bits as follows:

$$m = p_1p_2b_1p_4b_2b_3b_4p_8b_5b_6b_7b_8b_9b_{10}b_{11}$$

The encoded message contains 11 bits from the original message, plus 4 "parity bits." You probably have guessed that we're going to be creative about what goes into each parity bit. Here's roughly what happens.

Bit position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits	p_1	p_2	b_1	p_4	b_2	b_3	b_4	p_8	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}
p_1	●		●		●		●		●		●		●		●
p_2		●	●			●	●			●	●			●	●
p_4				●	●	●	●					●	●	●	●
p_8								●	●	●	●	●	●	●	●

There are many variants of this formulation. We'll use the following rule:

- The parity bits are called p_{2^i}, $i \geqslant 0$—that is, $p_1, p_2, p_4, p_8, p_{16}, \ldots$.
- Parity bit p_{2^i} is responsible for data bit index j for which `(1 << i) & j != 0`. In words, include index j in p_{2^i} if the i-th bit of index j written out in binary is 1.
- For example, what does p_2 equal?

3.1 On the receiving end...

Let's call the received message $r = r_1 r_2 \dots r_{15}$.

What should r look like if all is good? We should have, for example,

$$\begin{aligned}
\beta_1 &:= r_1 \oplus r_3 \oplus r_5 \oplus r_7 \oplus r_9 \oplus r_{11} \oplus r_{13} \oplus r_{15} &= 0 \\
\beta_2 &:= r_2 \oplus r_3 \oplus r_6 \oplus r_7 \oplus r_{10} \oplus r_{11} \oplus r_{14} \oplus r_{15} &= 0 \\
&\vdots \\
\beta_8 &:= r_8 \oplus r_9 \oplus r_{10} \oplus r_{11} \oplus r_{12} \oplus r_{13} \oplus r_{14} \oplus r_{15} &= 0
\end{aligned}$$

In the event that one of the bits is flipped, one of the β's will be nonzero. In fact, the following observation can be made:

> **Observation:** The exact bit that was flipped is $(\beta_8\beta_4\beta_2\beta_1)_2$. This value is 0 if no single bit was flipped.

Knowing which bit was flipped, we can easily correct the error by flipping that bit's value.

Detecting two errors? As it stands, we can't really distinguish between a single-bit flip and two-bit flip cases. However, this is easy to fix. Add a parity bit p_0, which is our good ol' friend: checksum.

At the expense of one more bit, we notice that we correct up to 1 bit flip, and detect up to 2-bit flip.

3.2 Analysis

The pattern seems to be clear: if the total message length is $2^r - 1$, we're going to have r parity bits and $2^r - r - 1$ real data bits. If we talk about percentage of real data transmitted, this would be captured by

$$R = \frac{\text{real data bits}}{\text{total length}} = \frac{2^r - r - 1}{2^r - 1}.$$

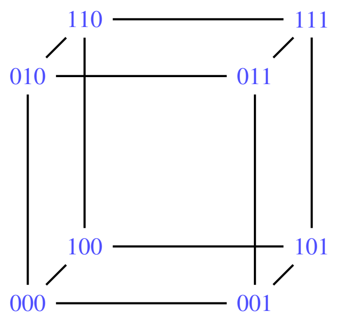
This is known in the literature as the rate of the code. For Hamming code, as we crank up r, we can see that we get a higher rate R.

r	Rate
2	0.333
3	0.571
4	0.733
5	0.839
6	0.905
7	0.945
8	0.969
9	0.982
10	0.99
11	0.995

4 Hamming is Optimal

How efficient is Hamming encoding? To show a lower bound on any code of this sort, we'll need a new tool.

Below is a picture of a hypercube Q_3. This Q_3 can be built from Q_2 recursively. In fact, Q_2 can be built from Q_1. To make Q_{n+1}, we first make two copies of Q_n. Call them $Q_n^{(0)}$ and $Q_n^{(1)}$. Prefix the label of each vertex in $Q_n^{(b)}$ with b.



Notice that every edge of this hypercube joins vertices that differ by exactly 1 bit. Therefore, if two vertices u and v differ by k bits, they are going to be connected by a path involving k edges.

Remember that we're focusing on a scheme that can recover from any 1-bit flip. A few observations can be made:

- If w is a codeword from an encoding scheme that can recover from any 1-bit flip, then every w' that differs from w by $\leqslant 1$ bit should be decoded as the original message.
  Say 000 is a codeword, then 100, 010, and 001 should all decode correctly to the same thing as 000. This leaves, 111, 011, 101, 110 to be decoded to the same thing.
- Therefore, two codewords (satisfying such a condition) must be at least distance 3 apart (in the hypercube).
- Every original message gives rise to a unique codeword.

From this observation, we can establish the following relationship between the number of codewords, the number of vertices in the hypercube (total possibilities of length ℓ bit strings).

- Because each original message gives rise to a unique codeword, we have that k-bit original messages correspond to a total of 2^k codewords (i.e., verticies).
- Each codeword w and every bit string w' that differs from w by one bit must decode to the same thing, so for each code word, $(n + 1)$ vertices decode to the same output.
- Hence, the total number of "used" bit strings are $2^k(n + 1)$, but this has to be at least the total number of bit strings on n bits. That is,
  $$2^k(n + 1) \leqslant 2^n$$

Taking log on both sides, we have $k + \log_2(n + 1) \leqslant n$. Using $n = 2^r - 1$, we have that $k \leqslant 2^r - 1 - \log_2(2^r - 1 + 1) = 2^r - r - 1$. Hence, any code of this nature (block size $2^r - 1$ that can recover from any 1-bit flip) can have rate at most

$$R \leqslant \frac{k}{n} \leqslant \frac{2^r - r - 1}{2^r - 1},$$

showing that the Hamming code described earlier is optimal in this sense.