Chapter 9 Error correcting codes.

Sections 9.1 and 9.2 are motivational mathematical problems that we will revisit as ideas are needed. We jump to §9.3.

Terminology for our discussion is as follows:

An alphabet will be some set of elements, like

$\{a, b, c, \ldots, x, y, z\}$

or $\{0, 1\}$

or $\{*, \#, ?, !\}$

and when we make a string out of elements of our alphabet, the result will be called a codeword.

E.g. The alphabet $\{0, 1\}$ results in binary codewords like 01001 01, 1111, 00, etc.

A set of codewords from our alphabet will be called a code.

E.g. The collection

$\{01001, 01, 1111, 00\}$

is a binary code.

If we further require that every codeword in the code must have the same length, we call it a fixed-length code.

E.g. $\{0011, 0101, 1111, 0000\}$ is a binary fixed-length code.

We're not going to discuss exotic alphabets like $\{*, \#, ?, !\}$ or even $\{a, b, c, \ldots, x, y, z\}$, because binary codes are sufficient to introduce all the concepts (our goal being to introduce Hamming codes as examples of error-correcting codes).

Last terminology: The Hamming distance between two binary codewords s and t is the number of positions where they differ. We write this as $d(s,t)$.

Remark: For $d(s,t)$ to be defined, s and t must have the same length. E.g. what would it even mean to count "the positions where they differ" if $s = 01$ and $t = 11010011$?

Example: $d(01011111, 01011001) = 2$, and

$d(111, 111) = 0$,

$d(010, 000) = 1$, etc.

Of course, we want to talk about the relationship between Hamming distance and codes, too, not just codewords.

The important concepts relating the two are:

The minimum distance of a code C is

$$d(C) = \min_{s,t \in C} d(s,t).$$

The maximum distance of a code C is

$$m(C) = \max_{s,t \in C} d(s,t).$$

E.g. Suppose C is the following fixed-length binary code:

$$C = \{001111, 010101, 000000, 010110\}.$$

Then $d(C) = d(010101, 010110) = 2$, since the pair $(0101\underbrace{01}, 0101\underbrace{10})$ differs the fewest times over all pairs.

only differences are here.

On the other hand

$m(C) = d(001111, 000000) = 4$, since the pair $(001111, 000000)$ has the most differences of all pairs of elements from C.

Our final bit of terminology is motivated by the usage of the code in question.

Suppose we transmit a message that is comprised of codewords from some fixed-length binary code.

For example, our message might be:

10101 01100 11010 10101 01100

comprised of codewords from the code $C = \{10101, 01100, 11010\}$. Now suppose that something goes wrong in our transmission of the message, and a few bits change. Say our friend receives the message:

10111 01100 11110 10101 01100

It would be reasonable for them to guess that transmission errors didn't occur too frequently - perhaps no more than once per code word. With that assumption, if they know the code C they can recover the original message! Why?

They know that the message is supposed to consist of codewords from C, so they know

10111 and 11110 are errors.

If they further assume only 1 error happened per codeword, then since 10101 is the only codeword in C with $d(10111, 10101) = 1$, they know 10111 was meant to be 10101. Similarly for 11110, they know it came from 11010.

The fact that they can detect and correct errors is because C has a special property:

If we change at most 2 digits in any codeword of C, then the resulting new codeword is *not* in C.

This is a really useful property! So we say a code C is *k error-detecting* if $s \in C$ and $0 < d(s,t) \leq k$ (where t is some string), then $t \notin C$. I.e. if you make a few changes to a codeword (but not more than k!) then the new string is not in C.

We say a code is *exactly* k-*error-detecting* if it is k-*error-detecting* and *not* (k+1)-error-detecting.

Examples

(1) Let C be the code consisting of all binary codewords of length 5 whose digits sum to an even number. Thus

$00000 \in C$ since $0+0+0+0+0$ is even

$10100 \in C$ since $1+0+1+0+0$ is even

$11100 \notin C$, etc.

There are $2^5 = 32$ strings of length 5. To count the elements of C, we only need to notice that choosing the first 4 bits determines the fifth:

If the first 4 bits are

1110 (for example)

then the last bit must be "1" for the result to lie in C.

Thus C contains $2^4 = 16$ elements.

- C is 1 error detecting, because changing a single bit in a codeword changes the parity of the sum of its digits.
- C is not 2 error detecting since

$$d(11000, 00000) = 2$$

yet 11000, 00000 are both in C.

(2) A less fancy example:

$$C = \{11111, 11100\}$$

is also an example of a code consisting of codewords of length 5 such that C is exactly 1-error detecting.

(3) Suppose C is all binary words of length 5, with the restriction that elements of C must have exact 2 ones. So

$$\underset{\uparrow}{1}\,0\,\underset{\uparrow}{1}\,0\,0 \in C$$

$$11100 \notin C$$

$$10000 \notin C \quad \ldots \text{etc.}$$

Then C has $\binom{5}{2} = \frac{5!}{2!(5-2)!} = \frac{5 \times 4}{2} = 10$ elements.

Again, this code is exactly 1 error-detecting.

§9.5 Error correcting.

In our earlier example of

$$C = \{10101, 01100, 11010\}$$

we observed that single-digit transmission errors could not only be detected, but also corrected since the Hamming distance of any two codewords in C is at least 3. This works in more generality:

We call a code C k-error-correcting if $d(s,t) \geq 2k+1$ for any pair of distinct codewords $s, t \in C$.

Example: The code C above is 1-error-correcting, and none of the examples from our discussion of error detection are also error correcting, because they were all 1-detecting. In fact, we have:

Theorem: A k-error-detecting code is $\lfloor \frac{k}{2} \rfloor$ error correcting.

Proof: If a code C is k-error detecting, then whenever $s, t \in C$ we have $d(s,t) \geq k+1$.

Note that if k is even, then $\lfloor \frac{k}{2} \rfloor = \frac{k}{2}$ and so $d(s,t) \geq k+1 = 2\lfloor \frac{k}{2} \rfloor + 1$, so C is $\lfloor \frac{k}{2} \rfloor$ error correcting.

If k is odd, then $\lfloor \frac{k}{2} \rfloor = \frac{k}{2} - \frac{1}{2}$, so $d(s,t) \geq k+1 = 2\lfloor \frac{k}{2} \rfloor + 2 \geq 2\lfloor \frac{k}{2} \rfloor + 1$, so again the code C is $\lfloor \frac{k}{2} \rfloor$ error correcting.

This is great, but there is still a shortfall to be corrected:

If our code is, for example,

$C = \{10101, 01100, 11010\}$

then it's 1-error correcting, which is great. However our message, if sufficiently corrupted, could easily end up containing codewords we don't know how to correct. E.g. if the message we receive is:

11101 01100 11010 10101 11111

Then we know there are errors in the first and last codewords. The first one, we correct it to 10101. The last one is distance 2 from two elements of C:

$d(10101, 11111) = 2, \quad d(11010, 11111) = 2$

so we don't know how to correct it.

Codes for which this ambiguity never happens do exist! They are called perfect, precisely:

A k-error-correcting code C is called perfect if

(i) C is a fixed-length code having codewords only of length n for some $n > k$, and

(ii) Every string of 0's and 1's of length n is within k of a unique element of C. I.e.

I.e. for each string t of length n there exists a unique $s \in C$ such that $d(s,t) \leq k$.

Hamming's (7, 4) code was the first perfect 1-error correcting code to be discovered. Our next goal is to describe his discovery, and other perfect codes.

§ Added content (not in text): A more detailed description of Hamming's (7, 4) code.

Hamming's (7, 4) code H is a fixed length binary code, all of whose codewords are of length 7.

A binary string $b_1 b_2 b_3 b_4 b_5 b_6 b_7$ is in H if and only if the bits $b_1, b_2, b_3, \ldots, b_7$ satisfy the following equations:

$$\begin{aligned} b_1 + b_3 + b_5 + b_7 &= 0 \\ b_2 + b_3 + b_6 + b_7 &= 0 \qquad \text{(all mod 2)} \\ b_4 + b_5 + b_6 + b_7 &= 0 \end{aligned}$$

Our next task is to describe why this code is perfect, and why it is 1-error correcting.

First, why is Hamming's (7,4) - code 1-error correcting?

Suppose that $b_1 b_2 b_3 b_4 b_5 b_6 b_7$ is a solution to the equations (*), ie it's a 7-bit string of 0's and 1's that is an element of Hamming's (7,4) code H.

First, note that if we change exactly one bit, the resulting string is no longer in H. This is because in the equations (*), each bit appears exactly once in a given equation (if it appears at all). So changing exactly one bit will change the parity of the left hand side, wherever it appears.

E.g. changing b_1 would change the parity of

$$b_1 + b_3 + b_5 + b_7$$

so it would no longer equal zero. Changing the value of b_5 would change the first and last equations, etc.

Next, observe that if we change two bits in $b_1 b_2 b_3 b_4 b_5 b_6 b_7$, the resulting string is not in H: To see this, observe that no matter what pair of bits (b_i, b_j) you choose, there is an equation in the collection of equations (*) that only contains one of b_i or b_j. So changing both b_i and b_j

again results in a parity change in some equation in ($*$), meaning the resulting string is not in H.

E.g. Suppose we change both b_3 and b_7. Then the terms b_3 and b_7 <u>both</u> appear in equations (1) and (2), so changing both b_3 and b_7 causes no problems in equations (1) and (2). However equation (3) contains only the term b_7, <u>not</u> b_3, so changing both b_3 and b_7 changes the parity of

$$b_4 + b_5 + b_6 + b_7$$

so that the resulting string is not in H.

Finally, observe that we can't go any farther with these types of arguments:

1010101 and

1111111

are both elements of H and $d(1010101, 1111111) = 3$.

This explains why H is 1-error correcting.

Why is H perfect?

To show this, we need to argue that for every 7-bit string that is <u>not</u> in H, there's exactly one (unique!) bit we need to change in order to arrive at a string in H.

So suppose

$B = b_1 b_2 b_3 b_4 b_5 b_6 b_7$ is not in H, and is an arbitrary 7-bit string. Recall the defining equations of H:

$$(1) \quad b_1 + b_3 + b_5 + b_7 = 0$$

$$(2) \quad b_2 + b_3 + b_6 + b_7 = 0$$

$$(3) \quad b_4 + b_5 + b_6 + b_7 = 0.$$

If B does not satisfy (1), we must change exactly bit 1, if it does not satisfy (2), we must change exactly bit 2, etc. The cases to consider are best summarized in a table:

Equatrons not satified by B	bit to change
(1)	1
(2)	2
(3)	4
(1) and (2)	3
(1) and (3)	5
(2) and (3)	6
(1) and (2) and (3)	7

It follows that H is perfect.

What is the Hamming code used for?

If you read online, you'll see the code described as something like "a method of transmitting four bits of data by encoding them in 7-bit strings, padded with parity checks".

To explain this:

Suppose you have an arbitrary 4-bit string you want to send. If we were restricted to using a fixed length code C containing strings of length 4, this would not be possible unless C contained all strings of length 4 - in which case C can't be error detecting or correcting.

So suppose instead we take our arbitrary 4-bit string and pad it; ie add another few bits:

$$\underbrace{b_1 b_2 b_3 b_4}_{\text{our string}} \; \underbrace{c_1 c_2 c_3}_{\text{padding}}$$

Then we can think of our 4-bit string (indeed, think of all 4-bit strings) as being members of a fixed length code containing elements of length bigger than 4.

E.g. We could think of the collection of all 4-bit strings as a subset of the fixed-length code

$C = \{ \text{all 6-bit strings } b_1 b_2 b_3 b_4 b_5 b_6 \text{ with } b_5 + b_6 = 0 \bmod 2 \}$

by sending $b_1 b_2 b_3 b_4$ (an arbitrary string) to

$$b_1 b_2 b_3 b_4 00 \in C.$$

We could also get more creative: We could think of our collection of 4-bit strings as a subcollection of 7-bit strings by sending

$b_1 b_2 b_3 b_4$ to $0 b_1 1 b_2 0 b_3 b_4$, or something similar.

* * *

So suppose our goal is to find a fixed length code C such that:

(1) C is error-correcting and perfect

(2) C contains all 4-bit strings, padded in some way to be longer

Then it turns out we need elements of C to be at least 7 bits in length - and Hamming's (7,4) code does the job!

For an arbitrary 4-bit string $b_1 b_2 b_3 b_4$, we can identify it with the element

$$c_1 c_2 b_1 c_3 b_2 b_3 b_4$$

where c_1, c_2, c_3 are bits that we compute as follows:

$$c_1 = b_1 + b_2 + b_4 \mod 2$$
$$c_2 = b_1 + b_3 + b_4 \mod 2$$
$$c_3 = b_2 + b_3 + b_4 \mod 2.$$

<u>Example</u>: The 4-bit string:

0111

gives
$$c_1 = 0 + 1 + 1 = 0 \mod 2$$
$$c_2 = 0 + 1 + 1 = 0 \mod 2$$
$$c_3 = 1 + 1 + 1 = 1 \mod 2$$

and so gets identified with the element

0001111

of Hamming's (7,4) code. <u>Ie.</u>: If we are using Hamming's (7,4) code to send a message, we would send 0001111 in place of the 4 bits 0111.

<u>Example</u>: We receive 1010101 from someone, and are communicating using Hamming's (7,4) code. This means their message was

$$1101 \quad \text{ie} \quad 10\overset{\downarrow}{1}0\overset{\downarrow}{1}\overset{\downarrow}{0}\overset{\downarrow}{1}$$

these bits.

Example: We receive the string

1011111 .

This is not in H, since it violates equation (2):

$$b_2 + b_3 + b_6 + b_7 = 0 \mod 2.$$

However it satisfies equations (1) and (3). So we know there has been an error in transmission, and it must be in the second bit! (Look back at the table of cases in our proof that H is perfect)

So the intended message is

1111111

which translates into the 4 bits

1111

ie, the sender intended 1111 as the message.

Other perfect error-correcting codes:

There are a few other examples of perfect error correcting codes, but we focused on Hamming's (7,4) code for a reason - it is by far the most tractable.

Another example is Golay's perfect 3-error correcting code. It is usually denoted G_{23}, and contains 4096 codewords of length 23.

Like Hamming (7,4), it is defined by a system of equations mod 2, but the system is enormous!

It is 11 equations, 23 variables, each equation containing 12 variables. Like Hamming (7,4), we can argue combinatorially from the defining equations that:

- $d(s,t) \geq 7$ whenever $s, t \in G_{23}$, so G_{23} is 3-error correcting
- If $s \notin G_{23}$ is a string of length 23, then there's a unique choice of at most 3 bits so that changing these bits of s results in an element $s' \in G_{23}$. Thus G_{23} is perfect.

We can further argue that G_{23} can be used to encode and send 12-bit strings - like using Hamming (7,4) to send 4-bit strings. However this argument requires us to actually write the defining equations of G_{23}. For this reason it is often denoted $G(23,12)$.