Intro to Error Correction

Alex S.*

1 Introduction

Error correction is accomplished by adding some redundancy to the transmission, which later (after transmission) lets us determine what the original message was (even if there were errors).

Theoretically, we can make any noisy channel reliable. The question becomes how much redundancy are we willing to add (and whether we can make such a decision before hand). For example, if we know what error rate to except, we can add enough redundancy to account for that error rate. If we don't know what the error rate is (cannot establish an upper bound), then whatever redundancy we add may not be enough.

1.1 Hamming Code (4,7)

For moderate error rates (and certain types of errors), we can define transformations that add redundancy. For example, we can take every 4 bits of the input, and turn them into 7 bits of output, using a scheme like this:

```
0000 => 0000000     1000 => 1000101
0001 => 0001011     1001 => 1001110
0010 => 0010111     1010 => 1010010
0011 => 0011100     1011 => 1011001
0100 => 0100110     1100 => 1100011
0101 => 0101101     1101 => 1101000
0110 => 0110001     1110 => 1110100
0111 => 0111010     1111 => 1111111
```

So a sequence like this:

```
1000 1010 1011 1110 1001 0110
```

will become:

*alex@theparticle.com

```
1000101 1010010 1011001 1110100 1001110 0110001
```

How exactly does that benefit us? We have to send lots of extra bits, etc. Well, for one, if we get the data that is one of the 7 bit patterns in the table, we know what was sent. For example, if we get: 0011100, we know that it transforms back to 0011, but if we get:

```
0000011
```

we know there was some error (it's not one of the patterns on the list). Can we use the received data to 'reconstruct' the sent data?

To reconstruct the data, we need to calculate the 'syndrome' of the received sequence. The syndrome is just a string representing the odd parity of some arrangement of the received bits. For example, we label the received bits from r1 to r7:

```
r1 r2 r3 r4 r5 r6 r7
 0  0  0  0  0  1  1
```

Then the syndrome is:

```
s1 = (r1 + r2 + r3 + r5) % 2;
s2 = (r2 + r3 + r4 + r6) % 2;
s3 = (r1 + r3 + r4 + r7) % 2;
```

Then, we flip a bit depending on what the syndrome is:

```
000 => don't flip any bits.
001 => flip bit r7
010 => flip bit r6
011 => flip bit r4
100 => flip bit r5
101 => flip bit r1
110 => flip bit r2
111 => flip bit r3
```

So if we received 0000011, the syndrome would be:

```
s1 = (0 + 0 + 0 + 0) % 2 = 0
s2 = (0 + 0 + 0 + 1) % 2 = 1
s3 = (0 + 0 + 0 + 1) % 2 = 1
```

Since syndrome is 011, we flip bit r4 of 0000011 to get 0001011, which just happens to be one of the patterns on the list, and nicely translates to: 0001.

1.2 Linear Codes

Hamming code is a linear code, and even though the above rules may seem kind of arbitrary, they can all be modeled using relatively precise mathematical notation. For example, the transformation is just a matrix multiplication (if you were implementing it, you would implement it as a table lookup). The decoding is also setup such that any single bit flip takes you back to a 'valid' pattern—so you're looking for the most likely bit flip (distance—number of bit flips). Again, if you were implementing it, you'd implement it as a table lookup.

These codes can be quite a bit more complicated; the Hamming 4,7 code is just the simplest for explanation.