Error–correcting codes with linear algebra

Jeff Jauregui

August 24, 2012

1 The problem

Consider a situation in which you must transmit a binary data signal (i.e., a sequence of 0's and 1's) across some channel. This happens in the use of WiFi, cell phones, lasers reading DVDs, and many other situations. In any realistic scenario, there is a possibility that errors will be introduced into the data stream: 0s will become 1s and vice versa in an unpredictable fashion. The problem is to devise a scheme by which errors can be detected and, ideally, corrected *automatically*.

Without such methods, our cell phone calls would not sound clear, our computers would crash more frequently, and CDs would sound scratchy.

2 Naive solutions

The simplest solution is to send each bit in segments of two: for each 0 in the intended message, you actually send two 0's, and similarly for 1's. Thus, the original message `[0 1 0]` would be encoded as `[0 0 : 1 1 : 0 0]`. We use : to divide up the string of bits into the segments. The receiver would, of course, need to be privy to this scheme.

The advantage to this approach is that the recipient can detect errors: if the message `[0 0 : 1 1 : 1 0]`, were received, there must be an error in the third segment of the message. The recipient can deduce that the *original* message must have been either `[0 1 0]` or `[0 1 1]`[1].

One disadvantage is that the recipient does not have enough information to *correct* the error. Another disadvantage is that twice as much data needs to be sent across the channel – this is problematic when bandwidth is a premium (e.g., you're on a limited data plan!).

The next most naive solution is to send each bit in segments of three. For example, encode the original message `[0 1 0]` as `[0 0 0 : 1 1 1 : 0 0 0]`. The downside is that three times as much data must be sent across the channel. The upside is that the recipient is now able to detect and correct errors. For instance, suppose the recipient

[1] From here on out, we will ignore the possibility of multiple errors occurring.

gets `[0 0 0 : 1 1 1 : 0 1 0]`. It should be clear that an error occurred in the last segment, since the sender only sends segments of `[0 0 0]` or `[1 1 1]` Now, in all likelihood, that last segment should have been `[0 0 0]`. Why? It's far more likely that the middle bit got switched from a 0 to 1 than the first and third bit both being switched from 1 to 0. Think of it like this: each bit in the triple is a vote for a 0 or a 1, and the majority rules.

2.1 Matrix formalism

As preparation for the next section, we give a matrix formalism to this last method of encoding. Warning: this methodology will probably seem unnecessarily complicated for now, but its utility will be evident later.

We define the *code generator matrix* G as

$$G = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Suppose $\vec{x}$ is one bit of the message you want to send, viewed as a 1×1 matrix. Then $G\vec{x}$ is a 3×1 column vector representing the encoded version of the sender's message:

$$G\vec{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \text{ if } \vec{x} = [1], \quad \text{and } G\vec{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \text{ if } \vec{x} = [0].$$

Next, suppose the message $G\vec{x}$ is sent, and the recipient gets a 3×1 vector $\vec{c}$:

$$\vec{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix},$$

where each of c_1, c_2, c_3 is 0 or 1. The hope is that $\vec{c}$ just equals $G\vec{x}$, but perhaps an error was introduced. The recipient looks for errors by means of a *parity check*[2]. For the message to be error free, you need $c_1 + c_2$ to be even (i.e. both are 0 or both 1), and also $c_2 + c_3$ to be even. (Question: why is it unnecessary to check that $c_1 + c_3$ is also even?)

To encode these requirements as a system of linear equations, **we will use "addition mod 2" from this point on**. This is described by the addition rules:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0, \end{aligned}$$

[2]The parity of a whole number is the property describing its even-ness or odd-ness.

which you can think of as "even plus even is even", "even plus odd is odd," etc. So our equations are

$$c_1 + c_2 = 0$$
$$c_2 + c_3 = 0$$

If you view c_1, c_2, and c_3 as unknown variables for this linear system of equations, the coefficient matrix is

$$P = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix},$$

which we call the *parity check matrix*.

Finally, the recipient takes $\vec{c}$ and computes the quantity $P\vec{c}$. There are four possible outcomes (where of course we are interpreting all numbers mod 2):

1. $P\vec{c} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. This means that $c_1 + c_2$ is even and $c_2 + c_3$ is even as well. In other words, no errors occurred, and $\vec{c}$ is the true message transmitted by the sender.

2. $P\vec{c} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. This means $c_1 + c_2$ is odd and $c_2 + c_3$ is even. So c_2 and c_3 are the same bit (both 0 or 1), and c_1 is different. Thus, the recipient assumes the error was in c_1, and so the intended message can be deduced.

3. $P\vec{c} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. **Exercise:** which bit is in error, and which bits are error-free?

4. $P\vec{c} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. **Exercise:** which bit is in error, and which bits are error-free?

Here's an interesting observation: the three possible outcomes of $P\vec{c}$ (other than the error-free case of $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$) are precisely the columns of P. In other words, in each of the three cases with an error, the number of the bit that is in error (first, second or third) is the same as the column number of P that $P\vec{c}$ equals.

To summarize this matrix formalism:

1. The sender takes a bit $\vec{x}$, encodes it as $G\vec{x}$ (where G is the 3×1 code generator matrix), and sends it across the channel.

2. The recipient gets some 3×1 message $\vec{c}$, and computes $P\vec{c}$, where P is the 2×3 parity check matrix.

3. If $P\vec{c}$ is the zero vector, then the message was free of errors. Otherwise, the recipient can readily determine where the error occurred and remedy it.

3 Hamming codes

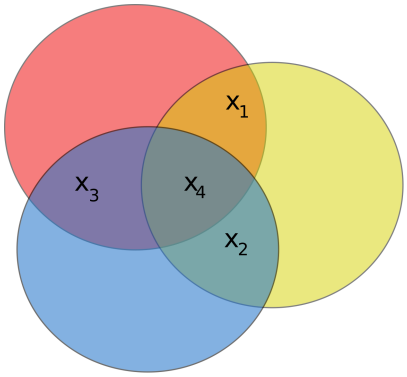
Hamming codes, developed by Richard Hamming in 1950, provide a method to send messages with error-detecting and -correcting, while using less than triple or even double the number of bits.

We focus on what is known as the "$(7, 4)$ Hamming code", which takes each group of four bits of the sender's message and encodes it as seven bits. Thus, this code requires that we send only $7/4 = 1.75$ times as many bits as the original message: much better than 3.

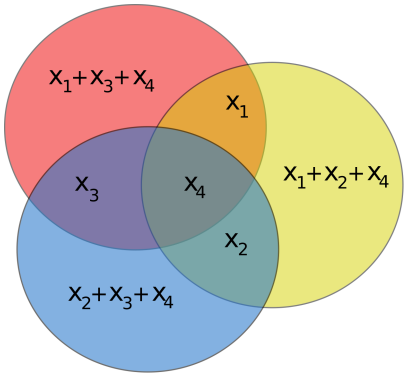
Suppose the message you wish to send consists of 4 bits, x_1, x_2, x_3, x_4. Denote this by the column vector

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}.$$

Perhaps the best way to describe the method is with a picture. In the following Venn diagram, write in the bits (0s or 1s) as follows. Think of the order as clockwise, spiralling inward[3].



Next, for each of the three circles, write the sum (mod 2) of the three bits that are located inside that circle, as follows.



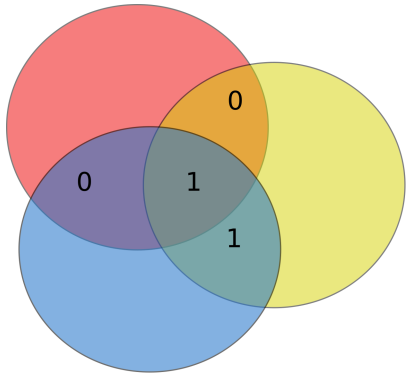
[3]The order in which we fill in these numbers is not important, as long as the sender and receiver remain consistent.

These three extra bits are called *parity bits*. The encoded version of the message will be the original four bits x_1, x_2, x_3, x_4 with these three parity bits tacked on to the end, for a total of seven bits. Fortunately, there's an easy way to construct this seven-bit string from the four-bit string using a new code generator matrix G:

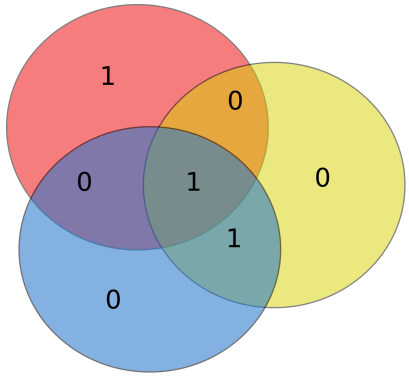
$$G = \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{array}\right].$$

The user sends $G\vec{x}$ (a 7×1 vector) in place of the original message $\vec{x}$. The matrix G is chosen precisely so that the first four entries of $G\vec{x}$ are x_1, x_2, x_3, x_4, and the last three entries are the parity bits $x_1 + x_3 + x_4, x_1 + x_2 + x_4$, and $x_2 + x_3 + x_4$. Let's see how this looks in an example.

Example. Suppose the message is `[0 1 0 1]` (written as a row vector only to save space). The initial picture is then:

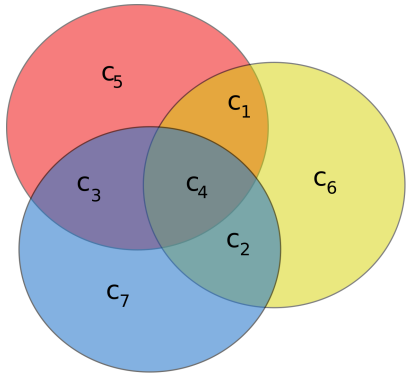


The parity bits get filled in as:



So in this case, the encoded message $G\vec{x}$ would be `[0 1 0 1 1 0 0]`. □

Back to the general case, the recipient receives some message of seven bits, call it $\vec{c}$. They *expect* the original message will just be the first four entries of $\vec{c}$, but they must check for errors. Graphically, the recipient draws the following picture...

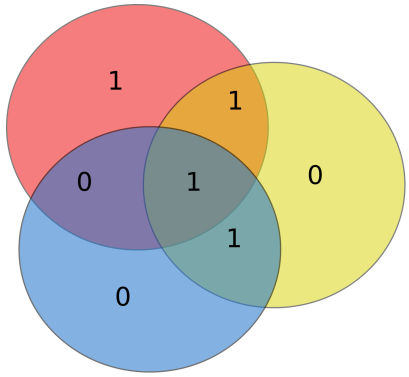


... and checks to see if it is valid. For instance, do c_1, c_2 and c_4 actually add up to c_6? Algebraically, the recipient is checking to see if the following three equations are satisfied (mod 2, of course):

$$c_1 + c_3 + c_4 + c_5 = 0$$
$$c_1 + c_2 + c_4 + c_6 = 0$$
$$c_2 + c_3 + c_4 + c_7 = 0.$$

Exercise: The coefficients of c_6, c_7, and c_8 should actually be -1 instead of $+1$. Why is it OK to use $+1$ in place of -1?

Example, cont. Continuing the previous example, suppose the recipient draws the following picture:



There seems to be a problem! The parity bit in the pink disk in the upper left is off: it should be 0, not 1. Similarly, the parity bit in the yellow disk on the right should be 1, not 0. This means an error occurred. Out of the seven bits that were transmitted, there is only one that could have caused this error: c_1, in the orange region between the pink and yellow circles. If the recipient changes c_1 from a 1 to 0, then everything checks out, and so the message must have been `[0 1 0 1]`. □

The coefficient matrix of the above linear system is

$$P = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix},$$

which we will again call the parity check matrix. Thus, the recipient computes $P\vec{c}$ (a 3×7 matrix times a 7×1 column vector), a 3×1 column vector. There are eight possible outcomes. First, if

$$P\vec{c} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix},$$

it means all the parity bits check out, so the message came through intact. As another example, suppose

$$P\vec{c} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

Each 1 in $P\vec{c}$ corresponds to a problem with the parity check. In this case, there is a problem with the parity bits c_5 and c_6. Looking at the picture from the last example, we see that there must be an error in the first bit c_1.

Thus, to recover the original message, the recipient takes `[c1 c2 c3 c4]` and swaps the c_1 (from a 0 to a 1 or a 1 to a 0). As yet another example, consider the case

$$P\vec{c} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

Where did the error occur? Well, it's only an error in the second parity bit, not with the message itself. So `[c1 c2 c3 c4]` is the true message.

Exercise: Examine the remaining five possible outcomes of $P\vec{c}$, and determine where the error occurred, and how to reconstruct the original message.

Exercise: If $P\vec{c}$ is not the zero vector, then it equals some column of P, say the ith column. Check that, in this case, the ith bit of $\vec{c}$ is the one in error.

Finally, to streamline this process, suppose you had a message of length $4k$ to send, where k is some positive integer. Record the bits in a $4 \times k$ matrix X, where you fill in the bits top to bottom, then left to right:

$$X = \begin{bmatrix} x_1 & x_5 & \dots & x_{4k-3} \\ x_2 & x_6 & \dots & x_{4k-2} \\ x_3 & x_7 & \dots & x_{4k-1} \\ x_4 & x_8 & \dots & x_{4k} \end{bmatrix}.$$

Encode this long message as GX (where G is still the same 7×4 code generator matrix), which is $7 \times k$ in size. The recipient gets a $7 \times k$ matrix C that may have picked up

some errors[4]. To check for errors, the recipient computes PC (which is $3 \times k$). If the ith column of PC is all zeros, then no error occurred in the ith column of the transmission. Otherwise, the recipient examines the ith column to determine where the error occurred, and corrects it.

4 Higher Hamming codes

Now that we've seen the Hamming $(7, 4)$ code that can correct errors at the expense of using 1.75 times the bandwidth, it is natural to ask if one can devise a code that uses an even smaller multiplier of the bandwidth. The answer is yes! The next simplest Hamming code is $(15, 11)$: it requires 15 totals bits for every 11 bits of the original message, for a multiplier of only $15/11 \approx 1.36$.

Challenge: Formulate the Hamming $(15, 11)$ code, which uses 4 parity bits. Pictures may be helpful. Find the code generator matrix and parity check matrix.

Final discussion: In theory, there are arbitrarily long Hamming codes of the form $(2^n - 1, 2^n - n - 1)$, where n is the number of parity bits. The multipliers get closer and closer to 1 (and therefore require only about the same amount of data as the original message), since

$$\lim_{n \to \infty} \frac{2^n - 1}{2^n - n - 1} = 1.$$

However, why would be it undesirable to make use of a code with n very large? See below[5] for a hint.

[4]Multiple errors are not an issue here, as long as at most one error occurs in each column of the transmitted message GX.

[5]Hint: The codes we discussed only work properly if at most one error occurs in each string.