Error Correcting Codes

Ben Meyers

December 2024

1 The Hamming Code

Hamming codes are a kind of error-correcting (the first!) codes that enable both detection and reversion of errors in binary data. If I wanted to send binary data over some network or wire, write it into a program, or store it in a disk or register, how can I protect myself from physical disruptions accidentally flipping a bit in the data? There are many different ways in which a bit, being represented somehow physically in transit or storage, can be flipped or erroneously shut to **0**. Data can become somewhat garbled in translation. Do I send it every couple seconds until I get a reply confirming the clean reception? What if that message has a bit-flip on its way to me? If storing on a disk, do I just give up if the message comes out messy? How can we embed within the data some mechanism to check and correct any errors that may arise in its transit or storage and do so efficiently?

1.1 Enter Richard Hamming and Claude Shannon:

While working at **Bell Laboratories**, these two pioneered **Error-Correcting Codes** in binary data that could first just detect errors, then eventually finding ways to precisely locate and thus fix errors in binary. This particular code, of course, was created by **Richard Hamming**, not **Shannon**, but sharing a desk at the Labs, they freely shared and built on each other's ideas, both pioneering aspects of information and communication theory. **Shannon and Hamming sought to answer a groundbreaking question:** ***Can we develop a method to encode binary messages in a way that not only detects errors but also corrects them efficiently?***

If we wanted to send or store, for example, the eleven-bit code [**10110101011**] , how could we embed within it a minimal set of additional bits to aide in error-correction? We could definitely imagine ways to stuff error-correcting **(EC)** bits between each data **(D)** bit and somehow use those to ensure fidelity, right? What if we, say, attached, for each bit in our message, two **EC** bits that are copies of the **D** bit that follows, so our message would become:

$$\begin{bmatrix} D & D & D & D & D & D & D & D & D & D & D \\ \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} \\ \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} & \text{EC} \end{bmatrix}$$

Or

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

and have the receiving machine just check in columnar trios and take the majority value? Could certainly work to add redundancy. But doing so triples the size of our message! At scale, this is a costly mechanism.

What about instead adding the binary sum at the end of the four-bit piece of data, so our example would become:

$$\begin{bmatrix} DDDDDDDDDDD\ \text{EC EC EC ...} \end{bmatrix}$$

Or

$$\begin{bmatrix} 10110101011\ 1\ 1\ 1 \end{bmatrix}$$

where the first four bits are the same old **D** bits and the three **EC** bits at the end correspond to the sum of the digits **(1 + 0 + 1 + 1 + 0 + 1 +0 + 1 + 0 + 1 + 1 = 7, or 111 in binary)**? What if instead we just had a single bit at the beginning or end that we set to either **1** or **0** depending on how to make the total sum of the bits (including the **EC** at the end) be even? Checking **sums** and **parity**, respectively, could both certainly tell us *whether or not* the data was messed with in any way. What about these check bits flipping, though? What about finding and remedying the error? Do we really want to have to retrieve another copy through another transmission if there is any error detected? Again, what about data storage? We need a way to not only detect, but fix!

1.2 Hamming's Elegant Encoding

As with many of the revolutionary insights at this time and place, reaching far and deep into the future, the answer lies in the powerful structure of binary digits themselves. The powerful and fundamental nature of a bit, *literally having only two possible values*, means that **finding** the location of an error means **fixing** it. We can use bits to represent so much more than numbers in their most plain form.

Building on the ideas of **sums** and **parity** discussed above, Hamming improved this concept in an incredibly intelligent way that allows the uncorrupted message to be recovered from one where a bit has been flipped. (To be clear, Hamming's code only work for a single bit flipping...it is however a conceptual jumping-off point and foundational for error correcting codes in general.) Using Hamming codes, for an eleven-bit message, we could instead add 4 **EC** bits, which we will now call Parity (**P**) bits to our message, carefully placed and assigned, and achieve some incredible results.

Whereas a parity check across the whole message leaves us blind as to where it may have originated, Hamming realized that with minimal overhead, you could divide the message into specific regions and insert **P** bits in certain places corresponding to these regions. In an originally eleven-bit message, we can add only **4** additional bits, thus creating **4** regions or groups on which to do separate parity checks, where the **P** bits that track parity are the first bit of each group.

Message with P bits inserted:

$$\begin{bmatrix} - & - & - & 1 \\ - & 0 & 1 & 1 \\ - & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \quad \text{Or} \quad [\ - - - 1 - 011 - 0101011\ ]$$

Indices of message:

$$\begin{bmatrix} 0000 & 0001 & 0010 & 0011 \\ 0100 & 0101 & 0110 & 0111 \\ 1000 & 1001 & 1010 & 1011 \\ 1100 & 1101 & 1110 & 1111 \end{bmatrix}$$

For the moment, try to ignore the very first bit, bit 0, as we will come to it later. For now, it is not a part of our message. Aside from a pretty pattern in the matrix view, why are we placing **P** bits where we are? Pause and ponder, specifically looking at the **indices**, why we may have placed these bits where we did.

If you notice in the indices, each **P** bit (which we will now call **P1, P2, P4, P8**) only has a single slot or *place* which contains a **1** in its binary string. Let us recall the mechanism of binary numbers before moving forward. We will count from **1** → **15** (the range of our indices of the new message):

1. **0001**
2. **0010**
3. **0011**
4. **0100**
5. **0101**
6. **0110**
7. **0111**
8. **1000**
9. **1001**
10. **1010**
11. **1011**
12. **1100**
13. **1101**
14. **1110**
15. **1111**

Recall that binary is simply a different way to represent quantities: is is **base-2**, meaning each *place* in a digit represents a quantity of powers of **2**, however many places away it is from the farthest on the right. Much like our conventional **base-10** system, where we have the **"1s place"** which denotes quantities of $\mathbf{10^0}$, and the **"2s place"** which denotes quantities of $\mathbf{10^1}$, and so on. The numeral **1011** in **base-10** represents $\mathbf{1*10^3+0*10^2+1*10^1+1*10^0}$ $=\mathbf{1000+10+1}$, which is, well, **1,011**. In binary, it represents $\mathbf{1*2^3+0*2^2}$ $\mathbf{+1*2^1+1*2^0=8+2+1}$, which is **11**. Do you see how dependent we are on **base-10** to even describe binary or **base-10**? Both of these methods of encoding numbers amount to **weighted sums** of distinct powers of a certain number. In a given encoding, say **base-X**, there are limits to how many you can "count" in a given **power of X**, specifically **X** many! To encode the number **11** in binary, we simply cannot say **51**, even though there are indeed $\mathbf{5*2^1+1*2^0=10+1}$. You might also note that the number of digits required to express a given number, **N** in a certain encoding, **base-X**, will always grow at a rate of $\mathbf{Log_X(N)}$. This will come in handy later as well and is really the definition of this kind of encoding; a short-hand way to talk about weighted sums with uniform and defined coefficients.

Base-10 is commonly thought to be a product of our own most primitive counting system, our **10** fingers. Binary, while not intuitive to us as a method of counting. It does, however, have the powerful property of restricting the possible values in a given digit to only **2** options. The key takeaways here are that **(A) if you know a binary digit is wrong, then you are guaranteed to know how to fix it precisely** and **(B) there is only one possible way to encode a given quantity in binary, a specific weighted sum of powers of 2, at most allowing 1 per power**.

Now we may return to the Hamming encoding and uncover why we placed the **P** bits as we did. Our **P** bits, **(P1, P2, P4, P8)**, are purposely assigned to the indices which are a clean **power of 2**, in other words, ones who's binary string has only a single **1**. Every single other index possible has at least **2 powers of 2** as a part of its weighted sum. If we assigned the location of these **P** bits according to this special property, then it makes sense we would carry it through and assign the groups as such too. Simply put, each **Parity** group is defined by indices having a **1** in a specific *place* in its binary string, by indices containing certain **powers of 2** in its weighted sum. Let's see how these groups shake out:

P1 (1, 3, 5, 7, 9, 11, 13, 15)

$$\begin{bmatrix} 0000 & 0001 & 0010 & 0011 \\ 0100 & 0101 & 0110 & 0111 \\ 1000 & 1001 & 1010 & 1011 \\ 1100 & 1101 & 1110 & 1111 \end{bmatrix}$$

P2 (2, 3, 6, 7, 10, 11, 14, 15)

$$\begin{bmatrix} 0000 & 0001 & 0010 & 0011 \\ 0100 & 0101 & 0110 & 0111 \\ 1000 & 1001 & 1010 & 1011 \\ 1100 & 1101 & 1110 & 1111 \end{bmatrix}$$

P4 (4, 5, 6, 7, 12, 13, 14, 15)

$$\begin{bmatrix} 0000 & 0001 & 0010 & 0011 \\ 0\boxed{1}00 & 0\boxed{1}01 & 0\boxed{1}10 & 0\boxed{1}11 \\ 1000 & 1001 & 1010 & 1011 \\ 1\boxed{1}00 & 1\boxed{1}01 & 1\boxed{1}10 & 1\boxed{1}11 \end{bmatrix}$$

P8 (8, 9, 10, 11, 12, 13, 14, 15)

$$\begin{bmatrix} 0000 & 0001 & 0010 & 0011 \\ 0100 & 0101 & 0110 & 0111 \\ \boxed{1}000 & \boxed{1}001 & \boxed{1}010 & \boxed{1}011 \\ \boxed{1}100 & \boxed{1}101 & \boxed{1}110 & \boxed{1}111 \end{bmatrix}$$

The more you look at these groups, the nicer you will feel. Let's explore how now to encode our message accordingly. To assign each **P** bit, we count the sum of their constituents, then assign the **P** bit, the first bit of the group, such that the total sum of the group is **even**.

Recall our pre-filled message:

$$\left[\ \boxed{-}\,\boxed{-}\,\boxed{-}\,1\,\boxed{-}\,011\,\boxed{-}\,0101011\ \right]$$

Parity calculations:

1. $P1+1+0+1+0+0+0+1 =$ **P1** $+$ $\boxed{3}$

2. $P2+1+1+1+1+0+1+1 =$ **P2** $+$ $\boxed{6}$

4. $P4+0+1+1+1+0+1+1 =$ **P4** $+$ $\boxed{5}$

8. $P8+0+1+0+1+0+1+1 =$ **P8** $+$ $\boxed{4}$

We see that **P1** and **P4** both currently have **odd** sums in their groups, so those bits must become **1** to make the group even, to give it **parity**. **P2** and **P8** may be assigned **0**.

Message with ordinary P bits *assigned*:

$$\begin{bmatrix} \boxed{-} & \boxed{1} & \boxed{0} & 1 \\ \boxed{1} & 0 & 1 & 1 \\ \boxed{0} & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \quad \text{Or} \quad \left[\ \boxed{-}\,\boxed{1}\,\boxed{0}\,1\,\boxed{1}\,011\,\boxed{0}\,0101011\ \right]$$

Now that we have Hamming-encoded our message, we are have only scratched the surface of the power of this system. The powers of 2 organize themselves elegantly in our matrix view, but beyond that, we will see that the meticulously chosen groups for parity will, with certainty, tell us where a bit was flipped if at all. Ponder again for yourself how we may go about decoding a message like this to reveal the original **11** bits, if a single bit is flipped...

Before we move to decoding, we must finally address the $\boxed{\mathbf{0}}$ location in our now length **16** string. We will see that the parity scheme we have now will lend itself perfectly to finding a single flipped bit, even a **P** bit, but we'd also like a way to know, even if we can't precisely fix it, *whether more that 1* bit was flipped. Of course the ideal scenario is no flipped bits, and we have prepared for **1** flipped bit, but it is still useful to know when we may need to abort this method entirely and address a $> \mathbf{1}$ flipped bit scenario. We will use this **0** index bit as an overall **P** bit for the whole message. To be clear, aside from this bit, while our message has groups with even **parity**, the overall message may still have odd **parity**. It turns out that ours does (**sum is 9**), so we will flip bit **0**, **P0**, to be **1**.

Final Hamming-encoded message:

$$\left[\ 1\ 1\ 0\ 1\ 1\ 011\ 0\ 0101011\ \right]$$

Or

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \quad \text{Or} \quad \left[\ 1101101100101011\ \right]$$

1.3 Decoding Hamming

Now we can dive into the decoding mechanism that highlights the genius of this scheme. It is trivial to see that we will be re-checking each of our **parity** to detect bit-flips, but just how it is done—giving the exact index of the erroneous bit—unearths an entirely novel way of using binary.

First, recognize the trivial case where no errors occurred in our message: each **parity** check will return **0**, as will the overall **P0** check. We will *know* that no errors occurred, because these were the **parity** conditions present after we encoded the message.

Now we will randomly flip a single bit in the message and find it. (For the purposes of illustration, I will show you which bit was flipped initially. You will still see the mechanism of detecting, locating, and fixing it all the same.)

Erred message:

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad \text{Or} \quad \left[\ 1101101100101\ 1\ 11\ \right]$$

First, we check the total **parity** of the message for **P0**:
$\mathbf{1+1+0+1+1+0+1+1+0+0+1+0+1+1+1+1=11}$, odd! . This is what we expected. We will see that a failed **P0** check means that at least **1** error has occurred—specifically an *odd* number of errors. If **2** errors had occurred, this **P0** bit would be blind to it, because it only refers to the whole message. This means that regardless of **P0**'s result, we must still make the rest of our **parity** checks:

P1: 1+1+0+1+0+0+1+1 = 5

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

P2: 0+1+1+1+1+0+1+1 = 6

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

P4: 1+0+1+1+1+1+1+1 = 7

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

P8: 0+0+1+0+1+1+1+1 = 5

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

Based on these **parity** checks, we can see that there was certainly an error. The checks that failed will tell us in which group the erroneous bit resides, right? It must be within **P2, P4,** or **P8**...but which one? One approach we could use is to find the intersection of all three of these sets, right? There is obvious overlap, but only one bit could possibly be in all these groups...right? Test it for yourself and see what your answer is.

By now you have seen that we have **2** options. We can see from **P1** that our index must be odd, in other words in the **2nd** or **4th** columns. From **P4**, we narrow our rows down to the **2nd** and **4th** as well. Finally **P8** narrows us further down to only that last row. Our answer must be an **odd** index in the last row...But there are two options! Index **13 (1101)** or **15 (1111)**. But which one?! We sifted exactly how it seemed would make sense, narrowing down the index by which **parity** checks it failed, yet we are still met with uncertainty. Other flipped bits may yield a certainty from this method, but here we see a counter-example. Our logic must be flawed in some way.

Taking a step back, let us talk through what failing the checks **P1, P4,** and **P8** really means numerically. It means our index in binary has:

1. a **1** in the $\mathbf{2^0}$ place
2. a **1** in the $\mathbf{2^2}$ place
3. a **1** in the $\mathbf{2^3}$ place

The index of our error is **1x11**. We know that **x** must either be **1** or **0**, thanks to the definition of binary. **1** would make **15** and **0** would make **13**. Now recognize a crucial detail: if the erroneous bit had been in the index **15**, then another check would have had to fail! **15** in binary is **1111**, so it belongs in every **parity** check that this message has, and thus would have to cause each of them to fail! Clearly, **P2** did not fail, so we can now say with certainty that the bit-flip occurred at index **13**, the **3rd** bit from the last.

Now, while this deduction was not too complicated, is there a way to run this decoding in one pass and not have to worry about uncertainty? From the method of deduction, it should be finding its way into your thoughts. A keen tinkerer might have first tried just adding up the **powers of 2** of each **parity** check that was failed, because, again, those are the groups in which we know the error occurred. Now this operation is very different from just finding the intersection between **3** sets. We saw that doing that could still leave ambiguity because of passed checks not being taken into account. It is not just that our error must reside in *all* of the **parity** groups that failed, it must reside in **only** those that failed. We can now see that:

$$\mathbf{P1} + \mathbf{P4} + \mathbf{P8} = \text{D13}$$

$$(\mathbf{1} * \mathbf{0001}) + (\mathbf{0} * \mathbf{0010}) + (\mathbf{1} * \mathbf{0100}) + (\mathbf{1} * \mathbf{1000}) = 1101$$

$$\mathbf{1} + \mathbf{4} + \mathbf{8} = 13$$

So it is not just that our **parity** checks alert us to regions where our error might be, *they spell out exactly where it is* without fail, and we know that with binary if you find an error, you have already fixed it. To appeal to those who want to feel the certainty of this method even deeper, I implore you to play with more examples of message lengths, different bits flipped in this message length, and what happens when more than **1** bit is flipped. Spoiler alert for the latter, **P0** will tell you exactly that. If an even number of bits are flipped, **P0**, will remain even, but other parity checks will inevitably fail. If an odd amount like **3** are are flipped, then when you add the **parities** that failed and then flip whatever bit you get, inevitably, other checks will still fail. This is how we can confidently judge with certainty **(A)** that **0** errors occurred, **(B)** that **1** error occurred and how to fix it, or **(C)** that more than **1** has been flipped and Hamming can no longer help us. As another base-covering measure, think of what happens when one of our **parity** bits is flipped...

1.4 One Final Complicated Simplification

For the savvy computer scientists reading this, you might feel a certain **Boolean** function nagging to be used in the back of your head. The all powerful **XOR**, exclusive **OR**, operation, denoted $\oplus$, can tell us whether or not an odd number of bits in a certain array are **1**. More narrowly, **XOR** can be used for **2** bits as input and only return **1** when *exclusively* one bit **or** the other is **1**. In an array, it simply tells us the **parity**.

How might you use the **XOR** operation to make the Hamming decoding process even more efficient without even having to manually check **parity** groups? Think of vertically aligning all of the indices in binary of the bits in our message that have a value of **1**. When encoded, if you take **XOR** on each *column*, you should expect **0000** as the answer, because, by construction, each **power of 2**, or **parity** group, has even **parity** when encoded. Now imagine what happens when you do this same operation on an erroneous message, like ours:

Hamming with XOR:

$$\begin{array}{rcccc}
 & 0 & 0 & 0 & 0 \\
 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 0 & 0 \\
 & 0 & 1 & 1 & 0 \\
 & 0 & 1 & 1 & 1 \\
 & 1 & 1 & 0 & 0 \\
 & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1} \\
 & 1 & 1 & 1 & 0 \\
 & 1 & 1 & 1 & 1 \\
\oplus & & & & \\
\hline
 & \boxed{1} & \boxed{1} & \boxed{0} & \boxed{1}
\end{array}$$

Why does this work? Look at what this result would be without our erroneous **1101** in the list. Whereas we started at all **0**s because we sent our message with parity, any additional **1** thrown into this mix will *modify* their respective columns where they have a **1**; again, in binary, you can only modify in one way.

2 The Hadamard Code

While Hamming codes provide exponentially efficient protection for **1** and detection for $>$ **2** errors in binary, the Hadamard/Walsh code allows you to protect a higher percentage of bits with certainty, if a bit more expensively. Hamming encoding allows for precise location of *one* error, but the Hadamard code groups possible **code words** by their similarity, specifically with respect to none other than ***Hamming* distance**. Hamming distance refers to the absolute sum of the differences between two binary strings. The maximum distance between any two strings of a certain length, **N**, is **N**. In Hadamard encoding, the maximum bit-flips tolerable in a string is $\mathbf{MaxDistance} = ((\mathbf{2^N})/\mathbf{4}) - \mathbf{1}$. In a string of length **4**, the maximum tolerable, we encode a **16**-bit message which can tolerate up to **3** errors. A **6**-bit message which gets encoded as a **64**-bit message which can tolerate up to **15** errors in transit or storage.

2.1 Encoding

To generate a Hadamard **codeword**, we will create a generator matrix, **H**, which has every possible **N**-bit input as columns. The number of columns, **K**, is always $\mathbf{2^N}$. We take the **inner-product modulo 2** of the input string with the columns of **H**, forming a length-**K** vector. If our vector is [**1001**], we will generate a $\mathbf{4} * \mathbf{16}$ **H**. In this matrix, note that the Hamming distance between any two rows is $\mathbf{K}/\mathbf{2}$, because each row is half **0** and half **1**. A single digit change in the input will cause half of the bits of its codeword to switch, because that bit will multiply to **1** in half of the columns exactly.

$$\overbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}}^{\mathbf{H}} \cdot \overbrace{\begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}}^{\mathbf{Input}}$$

$$\Downarrow$$

$$\overbrace{\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}}^{\mathbf{Codeword}}$$