Advanced Algorithms

Lecture 21: Coding Theory

Lecturer: Nathan Klein

1 Coding Theory

Consider the following puzzle sometimes known as the "hat problem."

> There are $n+1$ people standing in a circle. They are told that, shortly, someone will place hats on all but one of their heads. Each hat will be either red or blue. Furthermore, after the hats are placed on everyone's heads, they are told that there is no communication whatsoever allowed, with one exception: the one person without a hat will be allowed to say either "blue" or "red."
>
> Everyone in the circle can see the color of everyone else's hat, but not their own. So, before the proclamation, no one knows the color of their own hat. The question is: **what strategy maximizes the number of people who know the color of their own hat after the proclamation?**

One strategy would be, for example, for the person without the hat to tell the person on their left what color hat they're wearing. That certainly works, so at least one person will know. Another idea might be to say "blue" if there are more people with blue hats than red hats and "red" otherwise, but it's not clear this will even cause one person to know the color of their own hat.

Here's a strategy you might cook up to get two people to know: look on your left and right. If they are the same color, say "blue." This way both people next to you can check the hat color of the other and know their own color. But can you do better? It turns out, yes.

Fact 1.1. *There is a strategy that ensures everyone knows the color of their own hat.*

Proof. The person without a hat should say "red" if the number of people with blue hats is an even number, and "blue" otherwise. This way, each person can count the number of blue hats that they see. If they see an even number and "blue" was proclaimed, they know they must be blue, and if "red" was proclaimed they must be red. Similarly, if they see an odd number and "blue" was proclaimed, they must be red, whereas if "red" was proclaimed they must be blue. □

As it stands, this is just a cute puzzle. But if you haven't seen this before, hopefully you're appreciating how counter-intuitive it is that you can communicate to everyone, at the same time, the color of their hat with a single bit of information. It turns out this puzzle is the simplest example of a code.

1.1 What is Coding Theory

Coding theory is about designing ways to communicate information over noisy channels, or in other words, communicating in the real world where packets are dropped, transmissions are lost, hard drives fail, and you're not quite sure whether the person talking to you said "weeding" or "wedding."

So, suppose we have a message $x \in \{0,1\}^n$ which we want to send to someone. However, when we send it, some of the bits may be *erased* or *changed*, and we want the receiver to still be able to decode m. In our "weeding" or "wedding" example, and indeed in any exchange between two people, we can usually use context to fill in the missing or garbled word. Languages have redundancy baked into them. Our brains can correct "did you RSVP for the weeding" to "did you RSVP for the wedding" without skipping a beat (although a classy invite-only weeding sounds like fun).

Formally, let $\tilde{x}$ be the corrupted message received. We now want some way to recover the original message. This is impossible without adding extra bits to our message even if there's only one erasure (as we would be essentially compressing n bits into $n-1$ bits). So, in coding theory, we have a function $f : \{0,1\}^n \to \{0,1\}^m$ for some $m > n$. Given a message $x \in \{0,1\}^n$, we send $f(x)$. The goal is then to prove that under some error model, given $\tilde{x}$, we can recover $f(x)$.

In the hat problem, there are n bits, and it is possible that any single one of the n bits are erased, so that $x \in \{0,1\}^n$ where 0 is red, 1 is blue and $\tilde{x} \in \{0,1,*\}^n$ where $*$ indicates the erased bit. This corresponds to asking one of the n people (bits) what the color of their hat is (whether it's a 0 or a 1). To solve this, we encode $f(x) = (x, \|x\|_1 \pmod 2)$, i.e., x itself concatenated with a parity bit indicating whether the number of 1s (blue hats) is even or odd. This way we can recover the initial x. If the parity bit is erased, we return the first n bits of $\tilde{x}$, and otherwise we use the parity bit to fix the erased symbol.

1.2 Applications

A very important application of error correcting codes, and in particular erasure codes (which correct only erasures) is in data centers. Here, there are thousands of servers, and invariably at any time something like 1% of the servers will be down. And yet, data centers work: when you log into your email, your data is there with probability much higher than 99%.

The reason is that data is stored in a redundant fashion. A standard way to do this would be to make 3 copies of each data point, one on each of 3 different servers. This would increase the number of servers by a factor of 3. Now, you might argue it's not so likely that all the servers containing your data go down at the same time (although one should be careful when assuming server failures are independent as this is not always the case).

However, this is bad for two reasons. First, you multiply your storage by a factor of 3, which is very costly and should be avoided if possible. Second, this is not very robust. If just two servers go down, all the data stored on that triple of servers becomes unavailable. And even worse, in some settings the servers might be hard drives that can permanently fail. In this case, if three servers go down, you will permanently lose all data stored on that triple, which is a big issue: I would certainly stop using any service that in any given day had even a 0.01% chance of deleting all my data.

Instead, data is stored using error correcting codes. In this way, you can ensure any person's data can be retrieved even if a large number of servers go down. Even better, we don't need anywhere near a factor of 3 on the increase in number of servers. This is how data centers work at all major tech companies.

1.3 Some Formalism and the Singleton Bound

We just call the *range* of this encoding function f the code, which is some subset of $\{0,1\}^m$. The *distance* d of a code $C \subseteq \{0,1\}^m$ is:

$$d = \min_{x,y \in C} \|x - y\|_1$$

Fact 1.2. *If the distance of a code is d, then any $d-1$ erasures can be corrected and any $\lfloor \frac{d-1}{2} \rfloor$ errors can be corrected (in the information theoretic sense, ignoring algorithmic issues).*

Proof. Given any $\tilde{x} \in \{0,1,*\}^m$ with at most $d-1$ erased symbols, we need to show that there is a unique $c \in C$ such that $\tilde{x}$ could have been obtained by erasing $d-1$ symbols from c. So, suppose not and there are two codewords, c, c' with this property. But $\|c - c'\|_1 \geq d$. So they must differ on some symbol which was not erased: contradiction.

Second, consider any $\tilde{x} \in \{0,1\}^m$ with at most $\lfloor \frac{d-1}{2} \rfloor$ bits with errors. Suppose c, c' existed that are within $\lfloor \frac{d-1}{2} \rfloor$ errors of $\tilde{x}$, i.e. $\|\tilde{x} - c\|_1 \leq \lfloor \frac{d-1}{2} \rfloor$ and $\|\tilde{x} - c'\|_1 \leq \lfloor \frac{d-1}{2} \rfloor$. Then by the triangle inequality:

$$\|c - c'\|_1 \leq \|\tilde{x} - c\|_1 + \|\tilde{x} - c'\|_1 \leq d - 1$$

which is a contradiction. □

The code we constructed above has distance 2. This is because for any two input strings with distance 1, the parity bit will be different.

The **rate** of a code is $\frac{n}{m}$. Intuitively, this is the "information density" of your code: to send n bits, you need to blow up to m bits so every bit communicates (in a sense) $\frac{n}{m}$ bits.

There is a limit to how high the rate can be.

Fact 1.3 (Singleton Bound)**.** *For any code encoding n bits to m bits which can guarantee distance d, we must have $m \geq n + d - 1$.*

Proof. Consider the set $C \subseteq \{0,1\}^m$ of codewords. Now, consider the set C' which consists of removing the last $d-1$ bits from every codeword in C.

Since every pair of codewords has distance d, the remaining set C' still consists of all unique strings. But there are at most $2^{m-(d-1)} = 2^{m-d+1}$ of them. So, to ensure that every one of the 2^n possible messages is mapped to a unique codeword, we must have $m - d + 1 \geq n$, or rearranging, $m \geq n + d - 1$. □

Therefore, our construction with a single parity check is optimal, since here $d = 2$ and $m = n + 1$.

1.4 Linear Codes

There is a lot to say about coding theory, so this lecture is just the tip of the iceberg. But a very important primitive is *linear codes*. Here, we just mean that the encoding function is linear so that if $f : \{0,1\}^n \to \{0,1\}^m$ is the encoding function, we have $f(x+y) = f(x) + f(y)$ for all $x, y \in \{0,1\}^n$. So, to define a linear code, you just need to define $f(x)$ for all $x \in \{0,1\}^n$ with $\|x\|_1 = 1$, call them $e_1, \ldots, e_n$. Then to define $f(x)$, you can just sum up $f(e_i)$ for each index i with $x_i = 1$. Of course, for this to make sense (our output is in $\{0,1\}^m$, we need to take all entries mod 2.

Another way to think about this is we just need a matrix $G \in \{0,1\}^{n \times m}$ where row i is the codeword for e_i. G is called the *generating matrix* of the linear code. In this way, xG is the codeword for $x \in \{0,1\}^n$. (This is just convention, you can also look make the columns the codewords and look at Gx.)

It's not hard to see that without loss of generality, you can assume that G takes the form of the identity matrix on the first n columns. This is because you can perform elementary row operations without changing the set of codewords.

One very nice property of linear codes is that it is easy to understand the distance. To match the notation used in coding theory, for a binary string x we will call $\|x\|_1$ the Hamming weight of x.

Fact 1.4. *The distance d of a linear code is the smallest Hamming weight of any non-zero codeword in C.*

Proof. For any two codewords c, c', their distance is $\|c - c'\|_1$, which is itself a codeword since our encoding function is linear. (In particular if $f(x) = c, f(y) = c'$, then $f(x-y) = f(x+y) = f(x) + f(y) = f(x) - f(y) = c - c'$ since we are working over mod 2, or more precisely $\mathbb{F}_2$).

So, the distance is at least that of the minimum Hamming weight over all codewords $c \in C$. Furthermore this distance can be achieved, since $\mathbf{0}$ is a codeword. □

1.5 Hamming Code

So far, we have seen a code that corrects one erasure. Here we will see a code that corrects up to one error or two erasures: the $(7,4)$ Hamming code.

The generator matrix is the 4×4 identity, plus we will add 3 more columns, or "parity check" bits. These will consist of all seven 3-bit strings with Hamming weight at least 2:

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

We write $G = [I_n \mid P]$ to denote this matrix so that $G \in \{0,1\}^{n \times m}$.

What is the distance of this code? Using our above fact, it's the minimum Hamming weight of any non-zero codeword. A codeword is uniquely defined by which rows are added to produce it. Any single row has Hamming weight 3. If we add any three rows, we have Hamming weight at least 3 since we cannot remove the identity part. So, it remains to check the sum of two rows. Here, we will always get Hamming weight 2 from the identity part, and since no two parity check rows are the same, we will get an additional 1 from there. So we have distance 3.

Now we need to consider an algorithm for encoding and decoding. Encoding is easy, you just return xG. What about decoding? We know it is possible to decode, as we can just search for the nearest codeword, but the question is whether we can do so efficiently. For this, we form the

parity check matrix which is $H = \begin{bmatrix} P \\ I_{m-n} \end{bmatrix}$. Here that is:

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

Now the point is, $GH = P + P = 0$ (the matrix of all 0s) since every product can be broken up into the part you get from $I_n \cdot P$ and the part you get from $I_{m-n} \cdot P$.

Therefore, for $c = xG$, we have $cH = xGH = \mathbf{0}$. If the codeword we receive is not corrupted, we can uniquely decode. Otherwise, we may receive a corrupted codeword, $\tilde{c} = c + e_i$. Compute $\tilde{c}H$, which must be equal to $cH + e_iH$. We can simply read the row of H which corresponds to $\tilde{c}H$: this will indicate the corrupted bit. After flipping it, we can recover the original message.

While this $(7,4)$ code does not seem very useful, as it's restricted to input length 4, it can be made to work on any length input. To do so, you just break up your input x into "blocks" of length 4, and encode each of them using the (7,4) code.

However, we can construct Hamming codes for many sizes which get a much better rate. For any r, there is a Hamming code with input length $2^d - 1 - d$ and $2^d - 1$ length codewords with distance 3, so $f : \{0,1\}^{2^r - 1 - r} \to \{0,1\}^{2^r - 1}$. As r grows, this ratio goes to 1.

It turns out that for correcting a single error, Hamming codes are optimal.

1.6 Final Thoughts

We can go far beyond correcting one error (or two erasures). A beautiful linear code is the **Reed-Solomon** code. Here, we work over $\mathbb{F}_q$ instead of $\mathbb{F}_2$, and this lets us achieve the singleton bound: by adding just r additional symbols, you achieve distance $d + 1$, allowing us to correct $\lfloor \frac{r}{2} \rfloor$ errors or r erasures. The main drawback is the need to work over a large field. I will post some links to good resources on coding theory if you're interested.

Over $\mathbb{F}_2$, a good code to know is the LDPC code. This can be used to get a code with rate close to 1 and distance $\Omega(n)$, which is pretty impressive: it means you only need a small constant fraction of additional servers to be robust against a constant fraction of the servers going down.