Coding theory

Huy L. Nguyễn

In this note, we explore error-correcting codes, a solution for protecting against data corruption and data loss. To motivate the study, we start with a classical setup for storing data on a magnetic storage device considered by Hamming in 1950. We would like to store data in blocks of 63 bits. Over time, the data might become corrupted with bits flipping from 0 to 1 or vice versa. We assume that there is at most one such corruption per block. Thus, the major questions here are:

- How much information can we store reliably on each block of data?
- How to detect corruptions and correct them?
- Develop efficient algorithms to convert a message into an error-correctable form, called a *codeword*, and convert a (possibly corrupted) codeword back to the original message.

Repetition. One simple solution is repetition: we repeat the data 3 times. In this scheme, we use 63 bits to store a 21 bit message. To decode each bit of the message, we simply take the majority of the 3 repetitions. Why does this work? One explanation is that any two codewords differ on at least 3 bits. Therefore, after one corruption, there is at most one codeword within distance 1 from the received word and all other codewords are at distance at least 2 from the received word.

Observe that this solution can protect against many multi-bit error patterns e.g. one error in each block of 3 bits. However, it cannot protect against general two bit errors.

Checksum. To develop a more efficient code, we start with the task of detecting error. Notice that we can use a check sum to detect the error: that is, in addition to the message, we store 1 bit that is equal to the XOR of all bits in the message. Notice that the XOR of the bits of the codeword is always 0. Thus, we can use the XOR of the bits on the received word to check if it is corrupted or not.

Hamming code. In this scheme, we use 7 bits to store a 4 bit message b_1, b_2, b_3, b_4. The 3 additional bits store the checksums $b_1 \oplus b_2 \oplus b_4, b_1 \oplus b_3 \oplus b_4, b_2 \oplus b_3 \oplus b_4$. In other words, for a message x, the corresponding code word is xG, where the generator matrix G is

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

In order to show that this code can correct 1 mistake, it is sufficient to show that any two codewords differ in at least 3 coordinates.

Proof. Notice that any set of bits of a codeword along with their checksum satisfy the checksum condition we mentioned earlier (the XOR of the bits and the checksum is 0). Thus, for any codeword c, we have $Hc = 0$, where the parity check matrix H is

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

Notice that $HG^T = 0$ i.e. the product of the generator matrix and the parity check matrix is a matrix of zeroes.

Now we prove that any two codewords differ in at least 3 coordinates by contradiction. Consider two messages $x \neq y$ and suppose that the corresponding codewords $G^T x$ and $G^T y$ differ in at most 2 coordinates i.e. $G^T(x-y)$ is a binary vector with at most two non-zeroes. We have $HG^T(x-y) = 0$ because HG^T is the zero matrix. Notice that because $G^T(x-y)$ has at most 2 non-zeroes, the value of $HG^T(x-y)$ is the sum of at most 2 columns of H. All columns of H are distinct so it is not possible to add up two columns and get the zero vector. Thus, we have a contradiction. □

1 Random error and information theory

We next consider coding theory from a different point of view. In 1948, Shannon studied the problem of transmitting messages over a noisy channel. The simplest channel is where each bit is randomly flipped with some probability p. This is called the binary symmetric channel BSC_p. As we saw earlier, it is possible to correct for errors with redundancy. Shannon suggest the use of two algorithms for the coding process. First, we have an encoding algorithm $E : \{0,1\}^k \to \{0,1\}^n$ to transform a k bit message into an n bit codeword. The codeword is sent through the noisy channel and we receive a received word. Next, we have a decoding algorithm $D : \{0,1\}^n \to \{0,1\}^n$ to transform an n bit received word back to a k bit message.

The key question is how much redundancy is needed for each value of p. Shannon introduces the entropy function to answer this question.

$$H(p) = -p \log p - (1-p) \log(1-p)$$

Shannon shows that if $k/n < 1 - H(p) - \varepsilon$ then there exist E and D, and a constant δ such that $Pr[D(BSC_p(E(m))) \neq m] \leq \exp(-\delta n)$. On the contrary, if $k/n > 1 - H(p) + \varepsilon$ then there exists constant δ such that any E, D would fail with probability at least $1 - \exp(-\delta n)$. Here the probabilities are over the choice of m (uniformly random from $\{0,1\}^k$) and the noise of BSC_p.

Let's try to see where the entropy function comes from in the above theorem. Suppose we transmit a codeword $x_1, \ldots, x_n$ over the channel and receive $x_1 + y_1, \ldots, x_n + y_n$, where $y_i = 0$ with probability $1-p$ and $y_i = 1$ with probability p. How many flips i.e. how many 1s in y do we expect? By the Chernoff bound, the number of flips concentrates around $(1 \pm \varepsilon)pn$ with probability at least $1 - \exp(-\Omega(\varepsilon^2 n))$. Thus, a typical error pattern has around pn bit flips and there are $\binom{n}{pn \approx 2^{H(p)n}}$ such patterns. In order to be able to decode, we need to make sure that for any codeword and any common error pattern, the result is not confused with another combination of some codeword and a common error pattern. Thus, we have to allocate $2^{H(p)n}$ values to each codeword (one for each error pattern) and thus, can have at most $2^{(1-H(p))n}$ codewords.

2 Reed-Solomon codes

In this section, we consider the popular family of Reed-Solomon codes. These codes are used for many applications including storing data on CDs. We leave the setting of bits and assume that

each symbol takes one of q different values in the finite field Z_q (e.g. integers mod a prime q). We can add and multiply these values just like normal integers and $x \cdot y = 0$ if and only if either $x = 0$ or $y = 0$. A message of length k can be viewed as coefficients of a degree $k-1$ polynomial $p(x)$.

$$p(x) = m_0 + m_1 x + \cdots + m_{k-1} x^{k-1}$$

An important property of polynomials of degree $k-1$ is that any k evaluations uniquely determine the polynomial.

Lemma 2.1. *For any set of k pairs $(x_1, y_1), \ldots, (x_k, y_k)$ where x_i are distinct values in Z_q. There is a unique degree $k-1$ polynomial g such that $g(x_i) = y_i \ \forall i$.*

Proof. Let $m_0, \ldots, m_{k-1}$ be the coefficients of the polynomial. The constraints $g(x_i) = y_i$ are linear equations over these variables $m_0, \ldots, m_{k-1}$.

$$\begin{bmatrix} 1 & x_1^1 & \cdots & x_1^{k-1} \\ 1 & x_2^1 & \cdots & x_2^{k-1} \\ \cdots & & & \\ 1 & x_k^1 & \cdots & x_k^{k-1} \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ \cdots \\ m_{k-1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_k \end{bmatrix}$$

The left matrix is called the Vandermonde matrix. Its determinant is $\prod_{1 \le i < j \le k} (x_j - x_i) \neq 0$ because the x_i are distinct. Thus, there is a unique solution for this system of linear equations. □

The codeword is simply the evaluations of this polynomial at n points $u_1, \ldots, u_n \in Z_q$: $p(u_1), \ldots, p(u_n)$.

Suppose the channel corrupts r of these points where $r \le n - k$. Suppose the received values are $v_1, \ldots, v_n$. If we know where the corruption happens then we can recover the message by applying the above lemma to the evaluations that are not corrupted. Thus, the key issue is to figure out where the corruption happens. The following argument shows how to do so when $r \le (n-k)/2$.

Lemma 2.2. *There exists a degree r polynomial $e(x)$ and a polynomial $c(x)$ of degree at most $r + k - 1$ such that $c(u_i) = e(u_i) v_i$*

Proof. Let I be the set of indices that got corrupted. We choose $e(x) = \prod_{i \in I} (x - u_i)$ and $c(x) = e(x)p(x)$. Consider an index $i \in I$, we have $e(u_i) = 0$ and thus $c(u_i) = e(u_i)v_i = 0$. Consider an index $i \notin I$, we have $p(u_i) = v_i$ so $c(u_i) = e(u_i)v_i$. Thus, condition in the lemma holds for all i. □

The polynomial $e(x)$ is called the error locator polynomial. If we let the coefficients of c and e be unknowns then we have a system of $2r + k$ unknowns and $n \ge 2r + k$ equations. This system is overdetermined but the lemma guarantees that it is feasible. In fact, it is a system of linear equations so we can solve using Gaussian elimination.

We can define divisibility for polynomial: $e(x)$ divides $c(x)$ if there exists a polynomial $p(x)$ such that $e(x)p(x) = c(x)$. This is an analog of divisibility for integers and one algorithm for dividing polynomial is an analog of long division for integers.

Lemma 2.3. *If $n \ge 2r + k$ then any solution $c(x), e(x)$ satisfies 1) $e(x)$ divides $c(x)$ as polynomials and 2) $c(x)/e(x)$ is $p(x)$.*

Proof. Notice that $c(x) - e(x)p(x)$ has roots at all the u_i where the corresponding v_i is not corrupted. Thus, this polynomial of degree at most $r + k - 1$ has $n - r$ roots. Thus, if $n - r \ge r + k$ then this polynomial is identically 0. □

Thus, we obtain an algorithm for decoding Reed-Solomon code. First, we solve the linear system to find the coefficients of c and e. Then we use the long division algorithm to compute $c(x)/e(x)$, which gives $p(x)$.

3 Secret sharing

An interesting application of Reed-Solomon codes is to secret sharing due to Shamir. The model is as follows. Suppose we have a secret a_0 that we want to hide among n players. We would like to have the following 2 properties: 1) every subset of k people can pool their knowledge and decode it but 2) no subset of $k-1$ people can pool their knowledge and get any information about the secret.

Assume that a_0 is a number in Z_q. We pick random numbers $a_1, \ldots, a_{k-1}$ in Z_q and view $(a_0, \ldots, a_{k-1})$ as a message for Reed-Solomon code. Each player gets one symbol of the codeword corresponding to this message i.e. player i gets the evaluation of the polynomial $\sum_j a_j x^j$ at u_i. Because any k evaluation of a degree $k-1$ polynomial uniquely determine the polynomial, any k people can pool their knowledge and solve for the secret. However, for every set of $k-1$ people $i_1, \ldots, i_{k-1}$ and values $y_1, \ldots, y_{k-1}$ and a choice for a_0, there is a unique polynomial with the constant term a_0 and the evaluations matching their observations. Thus, from their point of view, any choice of a_0 is equally likely and thus they have no information about the correct value of a_0.

4 Multiparty secure computation

Next we consider a vast generalization of secret sharing called multiparty secure computation, due to Ben-Or, Goldwasser and Wigderson. Suppose each player i holds a secret s_i and the goal is to compute a function $f(s_1, \ldots, s_n)$, where f is a publicly known function. However, no subset of $k-1$ players can pool together their information and infer some information about other players' input beyond what they can compute from their inputs and the value $f(s_1, \ldots, s_n)$. For example, if the function f just reveals s_1 then we cannot keep s_1 secret from other players.

We assume that for any two players, there is a secure channel between them that can not be eavesdropped by other players. These channels can be constructed for example, by public key cryptography.

For the purpose of this note, we will only focus on a simple example: computing the sum of the secrets. We will also just focus on the case where all players are honest but curious: they try to learn as much as they can about others' secret but they will follow the protocol.

Each player runs a version of Shamir's secret sharing protocol. Player i picks $k-1$ random numbers $a_{i,1}, \ldots, a_{i,k-1}$ and evaluate the polynomial $s_i + a_{i,1}x + \cdots + a_{i,k-1}x^{k-1}$ at $u_1, \ldots, u_n$ and send the values to the respective players (the evaluation at u_i is kept to himself). Let $\gamma_{i,j}$ be the value sent by player i to player j.

At the end, we would like to compute $\sum_i s_i$. Now each player j compute $\sum_i \gamma_{i,j}$ i.e. sum of the shares sent to him. That is, he just pretends that the shares are the actually input data and perform the computation. A key observation is that this sum computed by player j is the evaluation at u_j of the polynomial

$$\sum_i (s_i + a_{i,1}x + \cdots + a_{i,k-1}x^{k-1})$$

This is a random polynomial and the value we would like to compute is the free coefficient of this polynomial. Thus, the player has managed to perform a secret sharing protocol for the sum $\sum_i s_i$.

We can also do a weighted sum of the secrets: the players compute the corresponding weighted sum of their shares.

It follows that we can also compute the product with a matrix $f(s_1, \ldots, s_n) = Ms$ since it is simply a sequence of weighted sums.

5 General computation

We would like to be able to compute securely any function, not just a simple sum. Just like boolean programs, we can define algebraic programs that capture all computation over a finite field.

A straight-line program of size m with inputs $x_1, \ldots, x_n \in Z_q$ is a sequence of m instructions of the form

$$y_i \leftarrow y_{i_1} \ op \ y_{i_2}$$

where $i_1, i_2 < i$ and op is either $+$ or $\times$ and $y_i = x_i$ for $i = 1, 2, \ldots, n$.The output of the program is y_m.

These programs can compute any polynomial in Z_q. Boolean straight line programs can perform arbitrary computation. A T step computation on a Turing machine can be simulated using a straight line program with $O(T \log T)$ steps.

The first n steps are easy, the players simply exchange shares. Consider the $n+1$ step. If it is a summation, we already saw how to compute a sum: each player computes the sum on their shares. Thus, the only operation left is to compute a product $y_i \leftarrow y_{i_1} \times y_{i_2}$. Suppose that these two previous variables are secretly shared using polynomials g and h where the value being secretly shared are the coefficients g_0 and h_0. The obvious way to secretly share the product is to use the product polynomial $\pi(x) = g(x)h(x) = \sum_{r=0}^{2k-2} x^r \sum_{i=0}^{r} g_i h_{r-i}$. The free coefficient $\pi_0 = g_0 h_0$ is exactly the desired product. Since everyone has evaluation of g and h, they can compute the evaluation of π by multiplying their evaluations.

However, there are a few issues. One problem is that this polynomial has degree $2k-2$ as opposed to $k-1$. The more serious problem is that the coefficients are not random elements of Z_q.

To make the coefficients random again, each player i picks a random degree $2k-2$ polynomial r_i with 0 as the free coefficient. He then secretly shares this polynomial with other players. Now the player can compute their secret shares for the polynomial

$$\pi(x) + \sum_{i=1}^{n} r_i(x)$$

Notice that the free coefficient of this polynomial is still $g_0 h_0$ and the other coefficients are uniformly random.

To truncate the degree of π at $k-1$, we can define a truncated polynomial $\pi'(x) = \sum_{i=0}^{k-1} \pi_i x^i$. It turns out that one can compute the evaluations of π' as the product of the evaluations of π and a fixed constant matrix. Details can be found in the original paper.