Proximity proofs

Thomas Coratger

1 An Introduction to Linear Codes

The journey into modern coding theory began in 1948 with Claude Shannon's groundbreaking paper, "A Mathematical Theory of Communication." He established that even over a noisy channel — like the vastness of space between a satellite and Earth, or a scratched compact disc — it's possible to transmit information with remarkable reliability. The key is to strategically add redundancy to the original message before sending it. This process, called **encoding**, transforms a message into a more robust form called a **codeword**. A **linear code** is a way to do this, using the structured language of linear algebra.

1.1 The Core Concepts

Before we build our first code, let's establish the fundamental concepts that allow us to measure information and errors.

1.1.1 The Alphabet, Codewords, and the Code Itself

First, we need an alphabet. In linear codes, this isn't just a set of symbols but a rich algebraic structure called a **finite field**, denoted $\mathbb{F}_q$. This is a set with q elements where we can perform addition, subtraction, multiplication, and division with well-defined rules.

Example: The Binary Field $\mathbb{F}_2$

The simplest alphabet is the binary field $\mathbb{F}_2 = \{0, 1\}$, where addition and multiplication are performed modulo 2. For instance, $1 + 1 = 0$.

With our alphabet, we can form strings. A **codeword** is a single vector (or string) of a fixed length, say n, composed of symbols from our alphabet. It represents one specific message after it has been encoded.

Example: A Binary Codeword

The vector $\mathbf{c} = (1, 0, 1, 1, 0)$ is a single codeword of length $n = 5$ in the space $\mathbb{F}_2^5$.

Finally, the **code** is the complete collection of all possible valid codewords. Think of it as a dictionary: each codeword is a valid entry, and the code, denoted $\mathcal{C}$, is the entire dictionary. The goal of decoding is to determine which "dictionary entry" was sent.

Example: A Simple Code

While $(1, 0, 1)$ is a single codeword, the set $\mathcal{C} = \{(0, 0, 0), (1, 0, 1), (0, 1, 1), (1, 1, 0)\}$ is the entire code.

1.1.2 Measuring Errors: Hamming Weight and Distance

To correct errors, we first need to quantify them. The Hamming metric provides an intuitive way to do this.

Hamming Weight The **Hamming weight** of a vector $\mathbf{u}$, denoted $w(\mathbf{u})$ or $|\mathbf{u}|_0$, is the number of its non-zero coordinates. It's a simple measure of how many positions are "active."

Example: Hamming Weight

For the vector $\mathbf{u} = (0, 1, 0, 1, 1) \in \mathbb{F}_2^5$, we count the number of ones. The Hamming weight is $w(\mathbf{u}) = 3$.

Hamming Distance The **Hamming distance** between two vectors $\mathbf{u}$ and $\mathbf{v}$, denoted $d(\mathbf{u}, \mathbf{v})$, is the number of positions in which their symbols differ. This is our fundamental measure of error — a distance of t means t symbols were flipped during transmission. For linear codes, there's a crucial link between distance and weight: the distance between two codewords is simply the weight of their difference: $d(\mathbf{u}, \mathbf{v}) = w(\mathbf{u} - \mathbf{v})$.

Example: Hamming Distance

Let $\mathbf{u} = (1, \mathbf{0}, 1, \mathbf{1})$ and $\mathbf{v} = (1, \mathbf{1}, 1, \mathbf{0})$. They differ in the 2nd and 4th positions, so their Hamming distance is $d(\mathbf{u}, \mathbf{v}) = 2$.

1.1.3 Minimum Distance

The single most important parameter determining a code's error-correction capability is its **minimum distance**, denoted ℓ. This is the smallest Hamming distance between any pair of distinct codewords in the entire code. A large minimum distance means the codewords are well-separated, making it harder for noise to change one codeword into another.

For linear codes, this simplifies beautifully: the minimum distance of the code is equal to the minimum Hamming weight of all its **non-zero** codewords.

$$\ell = \min_{\mathbf{u},\mathbf{v} \in \mathcal{C}, \mathbf{u} \neq \mathbf{v}} d(\mathbf{u}, \mathbf{v}) = \min_{\mathbf{w} \in \mathcal{C}, \mathbf{w} \neq \mathbf{0}} w(\mathbf{w})$$

This is because if $\mathbf{u}$ and $\mathbf{v}$ are distinct codewords, their difference $\mathbf{u} - \mathbf{v}$ is also a non-zero codeword (due to the linearity of the code). So, finding the minimum separation between all pairs is the same as finding the "lightest" non-zero codeword.

1.2 Formal Definition of a Linear Code

With these concepts, we can state the formal definition. An $[n, k, \ell]_q$ linear code $\mathcal{C}$ is a k-dimensional linear subspace of the vector space $\mathbb{F}_q^n$ with minimum distance ℓ.

- n: The **block length**. This is the total length of each codeword.
- k: The **dimension**. This is the number of original message symbols. Since $\mathcal{C}$ is a k-dimensional subspace over $\mathbb{F}_q$, it contains exactly $|\mathcal{C}| = q^k$ unique codewords. The ratio $\rho = k/n$ is the **rate** of the code, measuring how much of the transmitted data is useful information.
- ℓ: The **minimum distance**, which determines the error-correcting capability. A code with minimum distance ℓ can detect up to $\ell - 1$ errors and correct up to $t = \lfloor (\ell - 1)/2 \rfloor$ errors.
- q: The **size of the finite field** (the alphabet).

1.3 Constructing a Linear Code: The Generator Matrix

Because a linear code is a vector subspace, we can describe the entire set of q^k codewords using a small basis of just k vectors. We arrange these basis vectors as the rows of a $k \times n$ matrix called the **generator matrix**, G.

This provides a simple recipe for encoding: to encode a message vector $\mathbf{m} \in \mathbb{F}_q^k$, we compute the codeword $\mathbf{c}$ via matrix multiplication:

$$\mathbf{c} = \mathbf{m} \cdot G$$

The full code $\mathcal{C}$ is the set of all linear combinations of the rows of G.

Example: Constructing a $[3, 2, 2]_2$ Linear Code

Let's build a binary code using the following 2×3 generator matrix (we will see the details later on this generator matrix, the goal here is simply to see a first full example of a linear code):

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

Here, $n = 3$ and $k = 2$. We can generate all $2^k = 4$ codewords by multiplying each possible message vector $\mathbf{m} \in \mathbb{F}_2^2$ by G:

- For $\mathbf{m} = (0, 0)$: $\mathbf{c} = (0, 0) \cdot G = 0 \cdot (1, 0, 1) + 0 \cdot (0, 1, 1) = (\mathbf{0}, \mathbf{0}, \mathbf{0})$
- For $\mathbf{m} = (1, 0)$: $\mathbf{c} = (1, 0) \cdot G = 1 \cdot (1, 0, 1) + 0 \cdot (0, 1, 1) = (\mathbf{1}, \mathbf{0}, \mathbf{1})$
- For $\mathbf{m} = (0, 1)$: $\mathbf{c} = (0, 1) \cdot G = 0 \cdot (1, 0, 1) + 1 \cdot (0, 1, 1) = (\mathbf{0}, \mathbf{1}, \mathbf{1})$
- For $\mathbf{m} = (1, 1)$: $\mathbf{c} = (1, 1) \cdot G = 1 \cdot (1, 0, 1) + 1 \cdot (0, 1, 1) = (1 + 0, 0 + 1, 1 + 1) = (\mathbf{1}, \mathbf{1}, \mathbf{0})$

The complete code is $\mathcal{C} = \{(0, 0, 0), (1, 0, 1), (0, 1, 1), (1, 1, 0)\}$.
To find its minimum distance d, we find the weights of the non-zero codewords:

- $w(1, 0, 1) = 2$
- $w(0, 1, 1) = 2$
- $w(1, 1, 0) = 2$

The minimum non-zero weight is 2, so $\ell = 2$. This is a $[3, 2, 2]_2$ linear code.

1.4 How Good is a Code? The Singleton Bound

Now that we can build a code, a natural question arises: for a given length n, how much information (k) can we pack in while ensuring a certain error-correction capability (ℓ)? There is a fundamental trade-off, captured by the **Singleton Bound**.

For any $[n, k, \ell]_q$ code, the parameters must satisfy:

$$k \leq n - \ell + 1$$

This inequality sets a hard limit on the efficiency of any code. It tells us that we cannot arbitrarily increase both the dimension k (the amount of information) and the minimum distance ℓ (error-correction power) for a fixed block length n. Improving one often comes at the cost of the other.

1.4.1 Maximum Distance Separable (MDS) Codes

The most efficient codes are those that achieve equality in the Singleton Bound. A code is called a **Maximum Distance Separable (MDS) code** if its parameters satisfy:

$$k = n - d + 1$$

MDS codes are optimal because they offer the largest possible minimum distance for a given length and dimension, perfectly balancing information rate and error resilience.

Example: Checking for MDS Property

Let's check our $[3, 2, 2]_2$ example code against the Singleton Bound.

- The parameters are $n = 3$, $k = 2$, and $d = 2$.
- The bound states $k \leq n - d + 1$.
- Substituting the values: $2 \leq 3 - 2 + 1 \Rightarrow 2 \leq 2$.

The bound is met with equality! Our simple example code is an MDS code, meaning it's maximally efficient for its length and dimension.

1.5 The Encoding Process and Code Rate

The generator matrix provides a simple and powerful algebraic recipe for converting any message into its corresponding protected codeword. This process is called encoding. Here, we'll detail how it works and introduce the concept of code rate, a key measure of a code's efficiency.

1.5.1 The Encoding Map

The process of encoding is fundamentally a linear map from the k-dimensional space of messages to the n-dimensional space of codewords:

$$\text{encode} : \mathbb{F}_q^k \to \mathbb{F}_q^n$$

Given a message vector $\mathbf{m} = (m_1, m_2, \ldots, m_k) \in \mathbb{F}_q^k$, the encoding is performed by multiplying it by the $k \times n$ generator matrix G:

$$\mathbf{c} = \mathbf{m} \cdot G$$

The resulting codeword $\mathbf{c}$ is a vector of length n that now includes structured redundancy.

A particularly useful type of generator matrix is one in **standard form**, $G = [I_k|A]$, where I_k is the $k \times k$ identity matrix. An encoder using such a matrix is called a **systematic encoder**, because the original message $\mathbf{m}$ appears unchanged in the first k positions of the codeword $\mathbf{c}$. The remaining $n - k$ positions are the added redundancy, often called parity check symbols.

Example: Encoding a Single Message

Let's use our $[3, 2, 2]_2$ code from the previous example, which has the generator matrix:

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

Suppose we want to encode the message $\mathbf{m} = (1, 1) \in \mathbb{F}_2^2$. We compute the product $\mathbf{m} \cdot G$:

$$\begin{aligned} \mathbf{c} &= (1, 1) \cdot \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \\ &= (1 \cdot 1 + 1 \cdot 0, \quad 1 \cdot 0 + 1 \cdot 1, \quad 1 \cdot 1 + 1 \cdot 1) \\ &= (1, 1, 0) \end{aligned}$$

The message $(1, 1)$ is successfully encoded into the codeword $(1, 1, 0) \in \mathcal{C}$. Notice that since this G is not in standard form, the original message does not appear directly in the codeword.

1.5.2 Measuring Efficiency: The Code Rate

The **rate** of a code, denoted by ρ (or often R), measures its efficiency. It's the ratio of the message length to the codeword length:

$$\rho = \frac{k}{n}$$

The rate is a value between 0 and 1. It tells us what fraction of each transmitted symbol is actual information versus redundancy.

- A **high rate** (e.g., $\rho = 0.9$) means the code is very efficient, with little redundancy. This allows for faster data transmission but offers less protection against errors.
- A **low rate** (e.g., $\rho = 0.1$) means the code is less efficient, with a lot of redundancy. This slows down data transmission but provides much stronger error correction.

The choice of rate is always a trade-off between efficiency and reliability, dictated by the noise level of the communication channel.

Example: Comparing Code Rates

Let's compare two different codes.

1. A code $\mathcal{C}_1$ has parameters $[n = 10, k = 5]$. It encodes 5-symbol messages into 10-symbol codewords. Its rate is:
$$\rho_1 = \frac{k}{n} = \frac{5}{10} = 0.5$$
Exactly half of the transmitted data is information, and half is redundancy.
2. The famous $[23, 12, 7]_2$ Golay code, $\mathcal{C}_2$, encodes 12-bit messages into 23-bit codewords. Its rate is:
$$\rho_2 = \frac{k}{n} = \frac{12}{23} \approx 0.52$$
This code is slightly more efficient than $\mathcal{C}_1$ and is renowned for its powerful error-correcting capabilities, making it a classic example of a well-balanced code.

1.6 The Unique Decoding Regime

Now that we can construct and evaluate codes, we arrive at the central question of decoding: if we receive a corrupted vector $\mathbf{y}$, when can we determine the original codeword $\mathbf{c}$ with absolute certainty? This is the domain of **unique decoding**.

The key insight is that the minimum distance ℓ creates a "safe zone" or protective bubble around each codeword. As long as the number of errors is small enough that the received vector $\mathbf{y}$ remains inside the original codeword's bubble, we can reverse the damage unambiguously.

1.6.1 The Unique Decoding Radius

Because the minimum distance between any two distinct codewords is ℓ, their protective bubbles can have a radius of up to (but not including) $\ell/2$ before they risk touching or overlapping. This critical value defines the code's error-correction limit.

The maximum number of errors that a code is guaranteed to correct is denoted by t. It is defined as:
$$t = \left\lfloor \frac{\ell - 1}{2} \right\rfloor$$
Any received vector with t or fewer errors will always be closer to the original codeword than to any other. This value t is often called the **packing radius** or the **unique decoding radius** of the code.

The Uniqueness Guarantee Why can a received word only be "very close" to one codeword at most? The guarantee comes from a proof by contradiction that uses the triangle inequality.

Let's imagine for a moment that a received word $\mathbf{w}$ *is* very close to two different codewords, $\mathbf{u}$ and $\mathbf{v}$. To be "very close" means the distance is less than half the minimum separation, $\ell/2$. So we assume:

- The number of errors between $\mathbf{w}$ and $\mathbf{u}$ is small: $d(\mathbf{w}, \mathbf{u}) < \ell/2$.
- The number of errors between $\mathbf{w}$ and $\mathbf{v}$ is also small: $d(\mathbf{w}, \mathbf{v}) < \ell/2$.

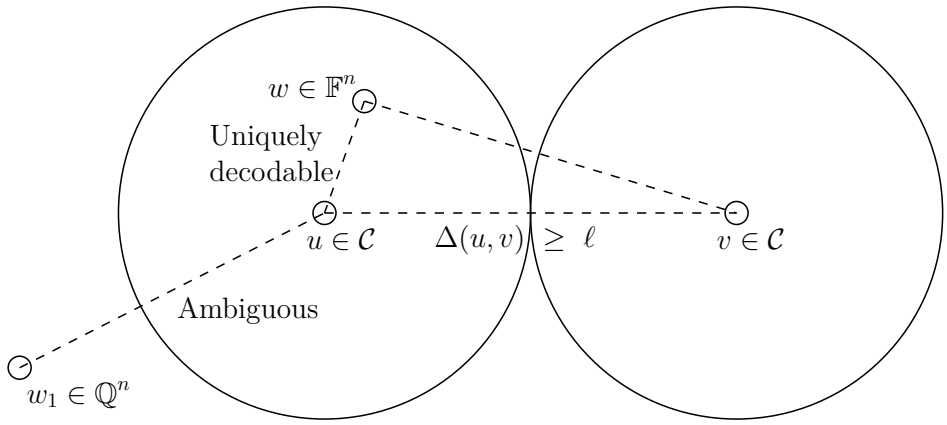

Figure 1: The unique decoding radius creates non-overlapping "safe zones" around codewords. A received $w \in \mathbb{F}^n$ inside a zone is uniquely decodable, while $w_1 \in \mathbb{F}^n$ outside is ambiguous.

The triangle inequality tells us that the direct path between two points is always the shortest. Going from $\mathbf{u}$ to $\mathbf{v}$ via the "detour" of $\mathbf{w}$ cannot be shorter:

$$\underbrace{d(\mathbf{u}, \mathbf{v})}_{\text{Direct distance}} \leq \underbrace{d(\mathbf{u}, \mathbf{w})}_{\text{Detour part 1}} + \underbrace{d(\mathbf{w}, \mathbf{v})}_{\text{Detour part 2}}$$

Now, let's see what our assumption implies. Since each part of the detour is shorter than $\ell/2$, their sum must be less than ℓ:

$$d(\mathbf{u}, \mathbf{v}) < \underbrace{\frac{\ell}{2}}_{\text{An upper bound for } d(\mathbf{u},\mathbf{w})} + \underbrace{\frac{\ell}{2}}_{\text{An upper bound for } d(\mathbf{v},\mathbf{w})} = \ell$$

This conclusion, $d(\mathbf{u}, \mathbf{v}) < \ell$, says that the distance between codewords $\mathbf{u}$ and $\mathbf{v}$ is less than the *minimum possible distance* ℓ. This is a logical impossibility.

Therefore, our initial assumption was wrong. A received word $\mathbf{w}$ cannot be simultaneously "very close" (unique decoding regime) to two different codewords.

Example: Unique Decoding in Action

Case 1: An Ambiguous Code ($t = 0$) Let's first revisit our $[3, 2, 2]_2$ code.

- Parameters: $n = 3$, $k = 2$, $\ell = 2$.
- The code is $\mathcal{C} = \{(0, 0, 0), (1, 0, 1), (0, 1, 1), (1, 1, 0)\}$.
- Unique decoding radius: $t = \lfloor (\ell - 1)/2 \rfloor = \lfloor (2 - 1)/2 \rfloor = 0$.

A radius of $t = 0$ means this code can **guarantee to correct 0 errors**. It can detect 1 error, but cannot reliably correct it. Suppose the received vector is $\mathbf{w} = (1, 0, 0)$. The number of errors is 1, which is greater than t. As expected, decoding is ambiguous:

- $d(\mathbf{w}, (0, 0, 0)) = 1$
- $d(\mathbf{w}, (1, 0, 1)) = 1$
- $d(\mathbf{w}, (1, 1, 0)) = 1$

The received vector $\mathbf{w}$ is equally close to three different codewords, so the decoder fails.

Case 2: A Uniquely Decodable Code ($t = 1$) Now, let's consider a code designed for error correction: the $[3, 1, 3]_2$ repetition code.

- Parameters: $n = 3$, $k = 1$, $\ell = 3$.
- The code is $\mathcal{C} = \{(0, 0, 0), (1, 1, 1)\}$.
- Unique decoding radius: $t = \lfloor (3 - 1)/2 \rfloor = 1$.

A radius of $t = 1$ means this code is **guaranteed to correct any single-bit error**.
Suppose we send $\mathbf{c} = (1, 1, 1)$ and a single error occurs, flipping the first bit. The received vector is $\mathbf{w} = (0, 1, 1)$. The number of errors is 1, which is within our decoding radius ($1 \leq t$). Let's find the closest codeword to $\mathbf{w}$:

- $d(\mathbf{w}, (0, 0, 0)) = w(0, 1, 1) = 2$
- $d(\mathbf{w}, (1, 1, 1)) = w(1, 0, 0) = 1$

The received vector $\mathbf{w}$ is uniquely closest to $(1, 1, 1)$. The decoder correctly identifies the original codeword, successfully correcting the error.

1.7 Why Unique Decoding Isn't Enough

Unique decoding provides a wonderful guarantee, but it comes with a strict limitation: it only works if the number of errors is very small. If the noise on a channel is higher, a received word $\mathbf{w}$ is highly likely to fall outside of any "safe zone." This leads to a crucial question: just how many words are actually uniquely decodable?

As we will see, the answer is a surprisingly small fraction. This limitation is the primary motivation for more advanced techniques like **list decoding**.

Counting the "Safe" Words To understand the problem, let's count how many vectors are in one of these uniquely decodable bubbles. A **Hamming ball** of radius t around a codeword $\mathbf{u}$, denoted $B(\mathbf{u}, t)$, is the set of all vectors $\mathbf{w}$ that are at a distance of t or less from $\mathbf{u}$.

The number of vectors in this ball is the sum of the counts for every possible number of errors, from 0 up to t. To understand the formula's origin, let's first figure out how to count the number of vectors with *exactly* i errors. This is a two-step process:

1. **Choose the error positions:** First, where did the errors occur? We need to choose which i of

the n total positions in the vector were corrupted. The number of ways to do this is given by the binomial coefficient $\binom{n}{i}$.

2. **Choose the new symbols:** For each of the i positions that have an error, the original symbol must be changed to a different one. Our alphabet is the field $\mathbb{F}_q$, which has q total symbols. This means for each error, there are $(q-1)$ other symbols to choose from. Since there are i errors, there are $(q-1)^i$ ways to choose the new symbols.

Therefore, the total number of vectors with exactly i errors is the product of these two parts: $\binom{n}{i}(q-1)^i$. To get the total size of the Hamming ball, we simply sum the counts for all possible numbers of errors, from $i=0$ (no errors) up to the maximum we can correct, t:

$$|B(\mathbf{u},t)| = \sum_{i=0}^{t} \underbrace{\binom{n}{i}}_{\text{Ways to choose } i \text{ error positions}} \times \underbrace{(q-1)^i}_{\text{Ways to choose new symbols for them}}$$

This formula counts every possible vector that can be reached from $\mathbf{u}$ by introducing at most t errors.

A Quick Refresher on the Binomial Coefficient $\binom{n}{k}$

The notation $\binom{n}{k}$, read as "n choose k", represents the number of ways to select k items from a set of n distinct items, where the order of selection doesn't matter.
The general formula is:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $n!$ (n-factorial) is $n \times (n-1) \times \cdots \times 1$. In our main formula, $\binom{n}{i}$ is simply counting all the possible ways that i errors can be distributed among the n positions of a codeword.

Example: Counting Words in a Hamming Ball

Let's use a code over the ternary field $\mathbb{F}_3$ (so $q=3$) with block length $n=10$. Suppose its minimum distance is $\ell = 7$, which gives a unique decoding radius of $t = \lfloor (7-1)/2 \rfloor = 3$.
How many words are in the Hamming ball of radius 3 around a single codeword $\mathbf{u}$?

$$|B(\mathbf{u},3)| = \sum_{i=0}^{3} \binom{10}{i}(3-1)^i = \binom{10}{0}2^0 + \binom{10}{1}2^1 + \binom{10}{2}2^2 + \binom{10}{3}2^3$$

Let's calculate term by term:

- 0 errors: $\binom{10}{0} \cdot 1 = 1 \cdot 1 = 1$ (just the codeword itself)
- 1 error: $\binom{10}{1} \cdot 2 = 10 \cdot 2 = 20$
- 2 errors: $\binom{10}{2} \cdot 4 = 45 \cdot 4 = 180$
- 3 errors: $\binom{10}{3} \cdot 8 = 120 \cdot 8 = 960$

The total number of words in this single decoding bubble is $1 + 20 + 180 + 960 = \mathbf{1,161}$.
Now, let's compare this to the total number of possible vectors in the entire space $\mathbb{F}_3^{10}$. There are $3^{10} = \mathbf{59,049}$ possible vectors.
So, the "safe zone" for a single codeword covers only $1,161/59,049 \approx 1.9\%$ of the entire space. Even if we sum this over all codewords in a high-rate code, the total uniquely decodable volume remains a tiny fraction of the whole space.

The example and the demonstration above reveal something important. The set of all uniquely decodable words (the union of all Hamming balls around all codewords) is almost always *exponentially smaller* than the total space of all possible received words, $\mathbb{F}_q^n$.

This means that if a received word $\mathbf{w}$ suffers from more than t errors, it will land in the vast and ambiguous space between the decoding bubbles. In this scenario, asking the decoder for a single, unique answer is asking the wrong question.

This is precisely why we need **list decoding**. Instead of demanding a single answer, we relax the question to: can you give me a short list of *all* codewords that are reasonably close to the received word?

This approach acknowledges that ambiguity is likely and provides a way to manage it, forming the basis for many modern and powerful decoding algorithms.

1.8 List Decoding

We've seen that the guarantee of unique decoding is powerful but fragile. It only applies within a small "safe zone" around each codeword. If a received word $\mathbf{w}$ suffers from more errors than the unique decoding radius t, it lands in the vast, ambiguous space where it might be close to several codewords at once.

Instead of giving up, we can change our goal. This leads to the idea of **list decoding**.

1.8.1 Motivation: A More Realistic Question

The problem with unique decoding is that it insists on a single, unambiguous answer. When that's not possible, it fails. List decoding takes a more different approach by asking a more flexible question: Instead of the *one* true codeword, can you give me a short *list* of all plausible candidates?

The goal is no longer to find a single answer but to efficiently produce a small list of all codewords $\mathbf{u} \in \mathcal{C}$ that are within a certain distance e of the received word $\mathbf{w}$.

The immediate question, however, is: how long could this list be? If it's potentially enormous, it's not very useful. This is where a remarkable result, the Johnson bound, provides a crucial guarantee.

1.8.2 The Johnson Bound: A Limit on Ambiguity

The **Johnson bound** is a universal "safety net" for decoding. It tells us that even when we are outside the unique decoding radius, the number of nearby codewords doesn't immediately explode. For a significant range of errors, the list of candidates remains small and manageable.

To state the bound formally, we use the *relative* parameters of the code:

- The **relative minimum distance**, $\mu = \ell/n$.
- The **relative decoding radius** (or fraction of errors), $\delta = e/n$, where e is the number of errors we want to tolerate.

The Johnson Bound

Let $\mathcal{C}$ be any code with relative minimum distance μ. For any received word $\mathbf{w}$, the number of codewords within a relative distance δ is bounded, as long as δ is less than the **Johnson radius**:

$$\delta < 1 - \sqrt{1 - \mu}$$

Inside this radius, the size of the list of candidate codewords is guaranteed to be small. Specifically, for binary codes ($q = 2$), the list size is bounded by approximately n. For general q, the bound is more complex but remains polynomial in n.

This is a powerful result because the bound depends only on the distance parameters μ and δ, not on the specific structure of the code.

The Johnson bound reveals that the ambiguity of decoding grows in a controlled way. It doesn't become chaotic as soon as we leave the unique decoding radius.

1.8.3 The Three Regimes of Decoding

We can now summarize the entire decoding landscape based on the fraction of errors, δ, we are trying to correct.

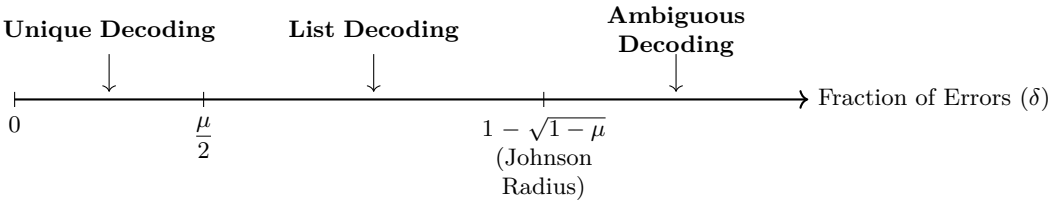

Figure 2: The three decoding regimes based on the fraction of errors δ.

1. **Unique Decoding Regime ($\delta < \mu/2$):** This is the "safe zone" we've already discussed. The list of candidates has at most one codeword.
2. **List Decoding Regime ($\mu/2 \leq \delta < 1 - \sqrt{1-\mu}$):** Here, we can no longer guarantee a unique answer, but the Johnson bound assures us that the list of potential candidates is small. Efficient algorithms exist that can find this list.
3. **Ambiguous Decoding Regime ($\delta \geq 1 - \sqrt{1-\mu}$):** Beyond the Johnson radius, all bets are off. The list of candidates could be exponentially large, and decoding becomes computationally infeasible for general codes.

The existence of the Johnson bound is crucial for modern coding theory. It proves that we can correct many more errors than the unique decoding radius suggests, as long as we are willing to accept a small list of answers instead of just one.

1.8.4 A Note on Terminology: δ-close and δ-far

To simplify our discussion of list decoding, it's helpful to have clear language to describe whether a received word is a plausible candidate for decoding or not. We use the terms **δ-close** and **δ-far** for this purpose.

Definitions

Let δ be a given fraction of errors (our relative decoding radius).

- We say a received word $\mathbf{w}$ is **δ-close** to the code $\mathcal{C}$ if there is *at least one* codeword $\mathbf{u} \in \mathcal{C}$ such that $d(\mathbf{w}, \mathbf{u}) \leq \delta n$. This means the list of potential candidates is not empty.
- We say a received word $\mathbf{w}$ is **δ-far** from the code $\mathcal{C}$ if *every* codeword $\mathbf{u} \in \mathcal{C}$ has a distance greater than δn from it, i.e., $d(\mathbf{w}, \mathbf{u}) > \delta n$. In this case, the list of candidates is empty, and a list decoder would return nothing.

This terminology provides a clean shorthand for discussing decoding problems. Instead of describing the conditions every time, we can simply ask: "Is the received word $\mathbf{w}$ δ-close to our code?" This captures the essential geometric question of whether $\mathbf{w}$ lies within the decoding radius of any codeword in $\mathcal{C}$.

2 Reed-Solomon Codes: Correcting Errors with Polynomials

Having established the fundamentals of linear codes, we now turn to one of the most powerful and widely used families of error-correcting codes in history: **Reed-Solomon (RS) codes**. Invented in 1960 by Irving S. Reed and Gustave Solomon, these codes are at the heart of technologies we use every day, from storing data on CDs and Blu-ray discs to ensuring the integrity of QR codes and transmitting data from deep space probes.

The genius of Reed-Solomon codes lies in a beautifully simple idea: **representing messages as polynomials**. By leveraging the unique properties of polynomials, we can add redundancy in a highly structured way, allowing us to correct not just single-bit flips, but entire chunks of corrupted data.

2.1 The Polynomial Toolkit

To understand RS codes, we first need to be comfortable with the tool they are built from: polynomials.

2.1.1 What is a Polynomial?

A polynomial is a mathematical expression consisting of variables and coefficients. For our purposes, a message polynomial $P(x)$ is a function of a variable x defined as:

$$P(x) = m_0 + m_1x + m_2x^2 + \cdots + m_{k-1}x^{k-1}$$

The values $m_0, m_1, \ldots, m_{k-1}$ are the **coefficients**, and the highest power of x with a non-zero coefficient is the **degree** of the polynomial. While we often think of coefficients and variables as real numbers, the real power of polynomials in coding theory comes from defining them over **finite fields**. This ensures all our calculations are exact and produce results that stay within our chosen alphabet.

2.1.2 The Fundamental Property: Uniqueness

The single most important property of polynomials for error correction is their uniqueness. A polynomial of degree $k-1$ is uniquely determined by any k of its points.

This means if you know the value of a degree-$k-1$ polynomial at k distinct points, you can perfectly reconstruct the one and only polynomial that passes through them. For example, two points define a unique line (degree 1), and three points define a unique parabola (degree 2).

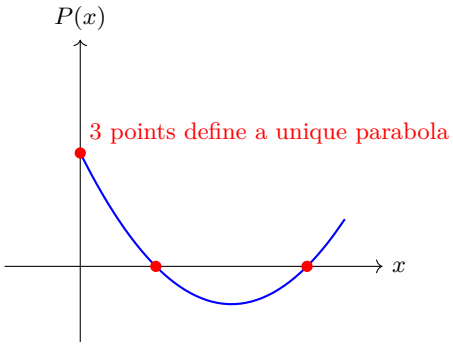

Figure 3: The uniqueness property of polynomials.

This property is what allows us to recover lost information. If a message is encoded as points on a polynomial, we only need to recover a sufficient number of those points (any k of them) to reconstruct the entire original message.

Example: Reconstructing a Polynomial

Suppose we are told that four points, $(-1, 2)$, $(0, 3)$, $(1, -5)$, and $(2, 1)$, lie on a polynomial of degree 3. Our message is encoded in the coefficients (m_0, m_1, m_2, m_3) of this polynomial:

$$P(x) = m_0 + m_1 x + m_2 x^2 + m_3 x^3$$

By plugging in each point, we get a system of four linear equations with four unknowns:

$$\begin{aligned} P(-1) = m_0 - m_1 + m_2 - m_3 &= 2 \\ P(0) = m_0 &= 3 \\ P(1) = m_0 + m_1 + m_2 + m_3 &= -5 \\ P(2) = m_0 + 2m_1 + 4m_2 + 8m_3 &= 1 \end{aligned}$$

Solving this system yields a unique solution for the coefficients: $(m_0, m_1, m_2, m_3) = (3, -4.5, -5.5, 2)$. The fractions highlight a key issue with using real numbers, which we'll address later using finite fields. For now, the crucial insight is that the four points uniquely locked in the polynomial and thus the message.

2.2 The Core Idea: Encoding by Oversampling

The most intuitive way to understand RS encoding is to view it as "oversampling" a function. We take a message, turn it into a polynomial, and then evaluate that polynomial at more points than are strictly necessary. These extra points are our redundancy.

This leads to a **systematic** encoding, where the original message is embedded directly within the final codeword. Here's the process:

1. **Interpret the Message as Points:** Take a message block of k symbols, $\mathbf{m} = (m_0, m_1, \ldots, m_{k-1})$. Treat these as the y-values of a polynomial $P(x)$ at k predefined evaluation points, say $x = 0, 1, \ldots, k-1$. This gives us k points: $(0, m_0), (1, m_1), \ldots, (k-1, m_{k-1})$.

2. **Interpolate to Find the Polynomial:** Find the unique polynomial $P(x)$ of degree at most $k-1$ that passes through these k points.

3. **Evaluate to Generate Redundancy:** "Oversample" the function by evaluating $P(x)$ at $n-k$ new points, say $x = k, k+1, \ldots, n-1$. These new values are our parity symbols.

4. **Form the Codeword:** The final codeword $\mathbf{c}$ is the list of all n evaluations. The first k symbols are the original message, and the next $n-k$ symbols are the redundancy.

Example: Systematic RS Encoding

Let's encode a message $\mathbf{m} = (3, 1)$ with $k = 2$ into a codeword of length $n = 4$ over a finite field (we'll use integers for simplicity, but imagine the math is modulo a prime, like 5).

1. **Message as Points:** We treat the message as the values of a polynomial $P(x)$ at $x = 0$ and $x = 1$. Our points are $(0, 3)$ and $(1, 1)$.

2. **Interpolate:** We need to find the unique line (degree $k - 1 = 1$) of the form $P(x) = a_0 + a_1 x$ that passes through these points.

   - $P(0) = a_0 = 3$
   - $P(1) = a_0 + a_1 = 1 \implies 3 + a_1 = 1 \implies a_1 = -2$

   So, our message polynomial is $P(x) = 3 - 2x$.

3. **Generate Redundancy:** We oversample by evaluating $P(x)$ at two new points, $x = 2$ and $x = 3$.

   - $P(2) = 3 - 2(2) = -1$
   - $P(3) = 3 - 2(3) = -3$

4. **Form the Codeword:** We concatenate the original message values with the new redundant values.

$$\mathbf{c} = (\underbrace{3, 1}_{\text{Message}}, \underbrace{-1, -3}_{\text{Redundancy}})$$

The final codeword is $(3, 1, -1, -3)$. If this is transmitted and one of the symbols is lost (an erasure), we can still recover the polynomial $P(x) = 3 - 2x$ from any of the remaining three points and regenerate the missing value.

2.3 Why Finite Fields are Essential

Throughout our examples, we've used integers for simplicity, but you may have noticed a problem: interpolation can produce fractions. Furthermore, the values of the evaluated points can become very large. Digital systems, however, work with fixed-size data types (like bytes) and cannot handle arbitrary fractions or infinite growth.

Finite fields (or Galois Fields, GF) solve both problems perfectly.

1. **No Fractions:** In a finite field, division is well-defined and always results in another element of the field. All calculations are exact.

2. **Closure:** The result of any addition or multiplication is always another element within the finite set of the field. This prevents numbers from growing too large and ensures that symbols always fit within their designated size (e.g., 8 bits for $\mathbb{F}_{2^8}$).

By constructing polynomials over a finite field like $\mathbb{F}_{256}$, Reed-Solomon codes become perfectly suited for modern digital storage and communication systems.

3 Proximity Gaps: An All-or-Nothing Property of Codes

We've explored how codes are built and how they can be used to correct a limited number of errors. But in advanced applications like modern proof systems (e.g., STARKs) and secure multi-party computation, we face a more complex challenge: verifying the integrity of not just one, but a whole *batch* of alleged codewords simultaneously. This leads to a profound structural question about codes, the answer to which has significant real-world consequences.

3.1 The Motivation: Verifying a Batch of Codewords

Imagine a scenario where a prover claims that a large set of vectors $\{\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_l\}$ are all valid (or at least very close to valid) Reed-Solomon codewords. Verifying each vector individually would be slow and inefficient. A much faster approach is to ask the prover to combine them into a single vector using random coefficients r_i:

$$\mathbf{u}' = r_0\mathbf{u}_0 + r_1\mathbf{u}_1 + \cdots + r_l\mathbf{u}_l$$

The verifier then only needs to check if this single random combination, $\mathbf{u}'$, is close to the code.

This raises a critical security question: **Is this test sound?** If a malicious prover provides just one vector $\mathbf{u}_i$ that is very far from the code, can they cleverly construct the other vectors such that their random combination $\mathbf{u}'$ almost always looks valid? If so, the test is useless.

For this test to be reliable, we need a guarantee that if even one vector is "far," the random combination will also be "far" with high probability. This is precisely what the theory of **proximity gaps** provides.

3.2 The Proximity Gap Phenomenon

A proximity gap is a powerful "all-or-nothing" principle. It states that for certain codes and structured collections of vectors (like lines or planes, known as affine spaces), there is no middle ground. Every vector in the collection is "close" to being a valid codeword, or almost none are.

Analogy: The Proximity Gap

Scenario 1: "All-In" (Correlated Errors) Imagine two slightly corrupted vectors, $\mathbf{u}_0$ and $\mathbf{u}_1$, where the errors happen to occur in the **same position**.

- Error in $\mathbf{u}_0$: $(0, 0, 5, 0)$
- Error in $\mathbf{u}_1$: $(0, 0, 2, 0)$

Any vector on the line between them, $\mathbf{u}_z = \mathbf{u}_0 + z \cdot \mathbf{u}_1$, will have a combined error of $(0, 0, 5 + 2z, 0)$. Notice that for any value of z, the error is still confined to a single position. If one error is correctable, they all are. Thus, **all** vectors on the line are "close" to the code.

Scenario 2: "All-Out" (Uncorrelated Errors) Now, imagine the errors are in **different positions**.

- Error in $\mathbf{u}_0$: $(5, 0, 0, 0)$
- Error in $\mathbf{u}_1$: $(0, 0, 2, 0)$

The combined error is $(5, 0, 2z, 0)$. Unless $z = 0$, this vector has two errors. If the code can only correct one error, then every vector on the line (except possibly $\mathbf{u}_0$ itself) is now "far" from the code. Thus, **almost none** of the vectors are "close".

The Insight: The rigid algebraic structure of polynomials prevents widespread, random cancellation of errors. The errors either align perfectly from the start (correlated), or they don't, in which case their combination almost always results in more errors. There is no scenario where roughly half the combinations are close and half are far. This is the essence of the proximity gap.

3.2.1 Formal Definition

This "all-or-nothing" idea is captured formally in the definition of a proximity gap.

Definition: Proximity Gap

A collection of sets $\mathcal{S}$ (e.g., all lines in $\mathbb{F}_q^n$) displays a (δ, ε)-proximity gap with respect to a code V if for every set $S \in \mathcal{S}$, exactly one of the following is true:

1. **All-In:** Every vector in S is δ-close to the code V.

$$\Pr_{\mathbf{u} \in S}[d(\mathbf{u}, V) \leq \delta n] = 1$$

2. **All-Out:** At most a tiny fraction, ε, of vectors in S are δ-close to the code V.

$$\Pr_{\mathbf{u} \in S}[d(\mathbf{u}, V) \leq \delta n] \leq \varepsilon$$

Here, δ is the proximity parameter (the fraction of allowed errors), and ε is the error parameter (a very small number).

3.3 The Main Result: Reed-Solomon Codes Have a Proximity Gap

The central theorem of the Ben-Sasson et al. paper[1] is that Reed-Solomon codes exhibit exactly this property. This provides the theoretical foundation needed to trust the random linear combination test.

The result is split into two regimes, depending on the fraction of errors δ we are willing to tolerate, compared to the code's rate $\rho = k/n$.

Theorem (Informal): Proximity Gap for RS Codes

The collection of all affine spaces has a strong proximity gap with respect to any Reed-Solomon code V. This holds for any fraction of errors δ up to the Johnson bound.

1. The Unique Decoding Regime ($\delta < \frac{1-\rho}{2}$) When the number of errors is small, the gap is extremely sharp. If an affine space is not "All-In", the fraction of close words is at most $\varepsilon = n/q$. For this to be a strong guarantee, we need the field size q to be larger than the block length n.

2. The List Decoding (Johnson) Regime ($\frac{1-\rho}{2} \leq \delta < 1 - \sqrt{\rho}$) Remarkably, the gap still holds for a much larger number of errors. The guarantee is slightly weaker, with the fraction of close words being at most $\varepsilon \approx O(n^2/q)$. To ensure this is a small number, we need a larger field, with q being roughly quadratic in n.

3.4 Application: Soundness of the FRI Protocol

The theory of proximity gaps has a direct and significant application in analyzing the security of modern proof systems. Specifically, it provides the mathematical foundation for the soundness of the FRI protocol[2] (Fast Reed-Solomon Interactive Oracle Proof of Proximity), which is a core component of transparent and quantum-resistant proof systems like STARKs.

3.4.1 The FRI Protocol: A Recursive Low-Degree Test

The primary function of the FRI protocol is to perform a **low-degree test**. In this context, a prover possesses a large dataset, represented as a function $f^{(0)}$ over a domain D_0, and asserts that this function is the evaluation of a low-degree polynomial $P_0(x)$ (i.e., that $f^{(0)}$ is a Reed-Solomon codeword). A verifier must be able to check this claim with high confidence by querying only a small number of points from the dataset.

FRI accomplishes this through a recursive process known as "folding." In each round, the prover reduces a given function to a new, smaller function, effectively halving its associated degree and domain size.

The FRI Folding Step

In each round i, the protocol begins with a function $f^{(i)}$ that is asserted to be of a certain low degree.

1. The prover splits $f^{(i)}$ into two components, often based on the even and odd coefficients of its polynomial representation:

$$f^{(i)}(x) = f^{(i)}_{\text{even}}(x^2) + x \cdot f^{(i)}_{\text{odd}}(x^2)$$

2. The verifier provides a random challenge scalar, $z_i \in \mathbb{F}_q$.

3. The prover uses this challenge to form a random linear combination of the two components. This defines the function for the next round:

$$f^{(i+1)}(y) = f^{(i)}_{\text{even}}(y) + z_i \cdot f^{(i)}_{\text{odd}}(y)$$

   The domain of $f^{(i+1)}$ is the set of squared elements from the previous domain, thus its size is halved.

This recursive process continues until the function is small enough for the verifier to check completely by reading all of its values.

3.4.2 The Soundness Argument

The security analysis of the FRI protocol involves two properties:

- **Completeness:** An honest prover, starting with a true low-degree polynomial $f^{(0)}$, will always pass the verification. This property is straightforward to show, as the folding process preserves the low-degree nature of the function at each step. This is often referred to as the "top-down" direction of the argument.
- **Soundness:** A dishonest prover, starting with a function $f^{(0)}$ that is far from any low-degree polynomial, will be caught with high probability. Proving this is more complex. It requires a "bottom-up" argument.

The core of the soundness proof is to show that the low-degree property propagates backwards up the recursive ladder. If the verifier is convinced that the final function $f^{(r)}$ is low-degree, it must follow that the initial function $f^{(0)}$ was also close to being low-degree.

3.4.3 The Role of the Proximity Gap in the Soundness Proof

The potential vulnerability in the soundness argument lies in the folding step. A malicious prover could attempt to construct functions $f^{(i)}_{\text{even}}$ and $f^{(i)}_{\text{odd}}$ that are both far from being low-degree, but whose random combination,

$$f^{(i+1)} = f^{(i)}_{\text{even}} + z_i \cdot f^{(i)}_{\text{odd}}$$

is close to a low-degree polynomial for a non-negligible fraction of the verifier's random challenges z_i.

This is precisely the scenario that the proximity gap theorem for Reed-Solomon codes demonstrates is not possible. The set of functions $\{f^{(i)}_{\text{even}} + z \cdot f^{(i)}_{\text{odd}} \mid z \in \mathbb{F}_q\}$ forms an affine line. The proximity gap theorem dictates that such a line must be either "All-In" (all functions are close to the code) or "All-Out" (almost no functions are close). A middle ground, where a significant fraction but not all functions are close, does not exist.

How the Proximity Gap Secures FRI

The proximity gap provides the critical logical step in the bottom-up soundness argument.

1. The verifier confirms that the final function, $f^{(r)}$, is a low-degree polynomial.

2. The verifier then proceeds backwards to round i. They are convinced that $f^{(i+1)}$ has high agreement with a low-degree polynomial (a codeword).

3. This means that for the prover's chosen $f^{(i)}_{\text{even}}$ and $f^{(i)}_{\text{odd}}$, their combination using the random challenge z_i was close to a codeword.

4. For the prover's attack to be successful, this would need to hold true for a significant fraction of possible challenges z_i.

5. The Proximity Gap Theorem states that if the line of functions $\{f^{(i)}_{\text{even}} + z \cdot f^{(i)}_{\text{odd}}\}$ is close to the code for more than a tiny fraction ε of challenges, it must be the "All-In" case.

6. The "All-In" case implies a strong structural property known as **correlated agreement**: the functions $f^{(i)}_{\text{even}}$ and $f^{(i)}_{\text{odd}}$ must themselves be close to low-degree polynomials, with their errors located in the same positions.

7. This correlated agreement is the necessary condition to prove that the parent function, $f^{(i)}$, was also close to being a low-degree polynomial.

By providing this inferential link at each recursive step, the proximity gap guarantees that the low-degree property propagates from the final function all the way back to the initial one. This ensures the soundness of the random linear combination check at the heart of FRI, which in turn allows the protocol to achieve high security with few queries, leading to smaller proof sizes and faster verification times.

3.4.4 From Tighter Soundness to Smaller Proofs

The primary goal of cryptographic proof systems like STARKs is to achieve a high level of security (e.g., 128 bits) as efficiently as possible. Efficiency is measured by the proof size and verifier computation time. The soundness analysis of FRI, backed by the proximity gap theorem, directly impacts this efficiency.

The connection is based on the relationship between security, queries, and proof size:

- **Proof Size depends on Queries:** A significant portion of a STARK proof consists of Merkle authentication paths that substantiate the prover's answers to the verifier's random queries. Therefore, the total proof size is directly proportional to the number of queries performed.
- **Queries depend on Soundness:** To reach a target security level, the verifier must perform enough queries to reduce the probability of accepting a false proof to a negligible amount (e.g., 2^{-128}). The number of queries needed depends on the *soundness error* of the protocol — that is, how effective each individual query is at catching a malicious prover.

This is where the proximity gap theorem provides a crucial improvement. Older analyses of the FRI protocol had looser bounds on the soundness error, suggesting that each query was only moderately effective at detecting fraud. To compensate, a large number of queries were required to achieve the desired security level, leading to larger proofs.

The proximity gap theorem provides a **tighter soundness analysis**. It proves that the random linear combination check at the heart of FRI is exceptionally reliable. This means each query is far more powerful at detecting inconsistencies than previously understood.

In summary, because the proximity gap provides a stronger guarantee on the soundness of FRI, fewer queries are needed to achieve the same level of cryptographic security. This directly translates to fewer Merkle paths in the final STARK proof, resulting in a smaller proof size and faster verification time.

References

[1] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf, "Proximity gaps for reed–solomon codes," *Journal of the ACM*, vol. 70, no. 5, pp. 1–57, 2023.

[2] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast reed-solomon interactive oracle proofs of proximity," in *45th international colloquium on automata, languages, and programming (icalp 2018)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 14–1.