NOTICIERO

ISSN 1514-9595 (web)

Miradas Matemáticas

Cuando se pierde un símbolo: codificando con inteligencia local

María Chara

Investigadora de CONICET en Universidad Nacional del Litoral

¿Por qué necesitamos códigos?

Imaginemos que queremos transmitir o almacenar información digital: una foto, un correo electrónico, un video o una base de datos. En todos estos casos, los datos viajan o se almacenan como secuencias de bits (ceros y unos), y siempre existe el riesgo de que alguno de estos bits se pierda o se altere. Puede ser por ruido en la transmisión, errores de lectura o fallas en un disco rígido.

Para enfrentar estos problemas, la **teoría de códigos** diseña métodos que permiten agregar *redundancia* a la información, de manera que se puedan detectar y corregir errores sin necesidad de repetir el mensaje entero.

Códigos lineales y sus parámetros

Uno de los modelos más estudiados es el de los **códigos lineales**. En términos generales, un código lineal es un subespacio vectorial $C \subseteq \mathbb{F}_q^n$, donde $\mathbb{F}_q$ es un cuerpo finito con q elementos (por ejemplo, $\mathbb{F}_2$, que sólo tiene los bits 0 y 1). Cada vector $\mathbf{c} \in C$ se llama *palabra código*, y representa una versión redundante del mensaje original.

El código se suele describir por tres parámetros:

- n: la longitud de cada palabra codificada (es decir, cuántos símbolos se transmiten o almacenan).
- k: la dimensión del subespacio C, que corresponde a la cantidad de símbolos de información original, (y por lo tanto hay q^k mensajes posibles para enviar).
- d: la *distancia mínima* del código, definida como la mínima cantidad de posiciones en que difieren dos palabras distintas del código. Formalmente,

$$d = \text{mín}\{\text{wt}(\mathbf{c}) : \mathbf{c} \in C,\ \mathbf{c} \neq \mathbf{0}\},$$

donde $\text{wt}(\mathbf{c})$ es el peso de Hamming, o cantidad de símbolos distintos de cero en $\mathbf{c}$.

Estos parámetros determinan la capacidad del código no sólo para enviar mensajes sino también para detectar y corregir errores.

Un código con distancia d puede:

- detectar hasta $d-1$ errores arbitrarios,
- corregir hasta $\left\lfloor \frac{d-1}{2} \right\rfloor$ errores,
- o bien recuperar hasta $d-1$ símbolos perdidos (borrados), si se sabe qué posiciones están afectadas.

Entonces, podríamos pensar que un *buen* código para una longitud n fija es aquel que tenga una dimensión grande (no queremos agregar muchísima redundancia) y una distancia mínima grande también, para poder detectar o corregir muchos errores. Sin embargo, estos parámetros están ligados entre sí por la famosa **cota de Singleton** que establece que

$$k + d \leq n + 1,$$

por lo que habrá que buscar un equilibrio.

Es importante notar que hay una diferencia entre *corregir un error* y *recuperar un símbolo perdido*. En el primer caso, el símbolo fue alterado pero no sabemos en qué posición ocurrió el error; en cambio, en el segundo, sabemos exactamente qué símbolo falta (por ejemplo, porque un servidor no respondió), y lo que queremos es *reconstruirlo* a partir de los demás.

Esta distinción es fundamental en contextos como el almacenamiento de datos, donde lo más común no son errores arbitrarios, sino *borrados*: bloques de datos que simplemente desaparecen o no están disponibles temporalmente. Los códigos lineales permiten enfrentar ambos problemas, pero la recuperación de borrados suele ser más eficiente, ya que contamos con información adicional sobre la ubicación del problema.

Un ejemplo autocorrector

Veamos un ejemplo sencillo que ilustra estas ideas. Supongamos que queremos transmitir dos bits a y b en $\mathbb{F}_2$, y usamos el código:

$$(a, b, a+b),$$

donde la suma es módulo 2. Este es un código lineal con parámetros $[3, 2, 2]$, ya que hay dos bits de información, cada palabra tiene tres símbolos, y cualquier par de palabras difiere en al menos dos posiciones.

¿Qué permite este código? En primer lugar, puede *detectar un error*: si recibimos los tres símbolos pero la suma de los dos primeros no coincide con el tercero, sabemos que al menos uno fue alterado. Por ejemplo, si recibimos $(1, 0, 0)$, la suma $1 + 0 = 1$ no coincide con el tercer símbolo, así que hubo un error.

Además, si uno de los tres símbolos se pierde (es decir, se borra), podemos recuperarlo conociendo los otros dos. Por ejemplo, si recibimos $(*, b, a+b)$, sabemos que el símbolo faltante es $a = (a+b) + b$. De forma similar, podemos recuperar cualquiera de los tres si conocemos los otros dos.

Este es un ejemplo minimalista de un **código corrector de errores**. Sin embargo, en códigos más largos, puede ocurrir que para recuperar un solo símbolo perdido tengamos que acceder a *todos* los demás. ¿Es eso realmente necesario? ¿O podemos diseñar códigos donde cada símbolo se pueda recuperar de manera más eficiente?

El problema de la recuperación local

En contextos como el almacenamiento en la nube, donde los datos están repartidos entre miles de servidores, acceder a todos los símbolos para recuperar uno solo puede ser muy costoso. Por eso surge la pregunta natural:

> *¿Podemos diseñar códigos que permitan recuperar cada símbolo accediendo sólo a una pequeña cantidad de los demás?*

La respuesta es sí, y esos códigos se llaman **códigos con recuperación local** (LRC por sus siglas en inglés). Formalmente, un código tiene *recuperación local* r si cada símbolo del código puede recuperarse a partir de a lo sumo r otros símbolos. Idealmente, r es mucho menor que la longitud total n del código, y más aún, que la dimensión k del código. De hecho, si accedemos a k símbolos linealmente independientes podemos reconstruir toda la palabra codificada, pero eso sería innecesariamente costoso si lo único que queremos es recuperar un símbolo perdido.

Una propiedad interesante es que muchos de estos códigos permiten no una, sino varias formas disjuntas de recuperar un mismo símbolo. Esta característica, conocida como *disponibilidad*, resulta muy útil cuando hay múltiples fallas simultáneas: si un conjunto de recuperación está dañado, podemos usar otro. Formalmente, decimos que un código tiene *recuperación local con disponibilidad* t si cada símbolo puede recuperarse a partir de al menos t conjuntos disjuntos de otros símbolos, cada uno de tamaño a lo sumo r.

Un caso práctico: una nube con memoria

Imaginemos que subimos nuestras fotos a un sistema de almacenamiento distribuido. El sistema divide nuestras fotos en partes, y luego codifica cada conjunto de partes usando un LRC. Así, aunque un servidor deje de responder o un disco falle, el sistema puede reconstruir cada pieza perdida usando solo algunas pocas de las otras, sin necesidad de mover grandes cantidades de datos. Este enfoque es especialmente importante en empresas que ofrecen almacenamiento en la nube, donde el tráfico interno y la eficiencia son críticos.

En efecto, Microsoft implementó un sistema de archivos llamado Azure Storage basado en códigos con recuperación local. Según documentación de Microsoft Research, su implementación de códigos LRC permitió reducir el número de fragmentos leídos durante la recuperación —de 12 a 6 en el caso más común— en comparación con un sistema basado en Reed–Solomon. Esto se tradujo en más del 50 % de ahorro en ancho de banda sin comprometer la confiabilidad (ver [2]).

Una construcción algebraica: el código de Tamo y Barg

Construir códigos con recuperación local no es trivial, y hay que tener en cuenta que queremos garantizar que el código tenga buena *distancia mínima*, para poder corregir

errores; que cada símbolo tenga un conjunto de recuperación de tamaño acotado y menor a la dimensión; y que la codificación y decodificación sean eficientes.

Una de las construcciones más elegantes y generales es la que propusieron **Tamo y Barg** ([1]) en 2014. Su idea central es definir el código como un *código de evaluación*, es decir, codificar un mensaje como los valores de un polinomio en ciertos puntos de un cuerpo finito. Pero no cualquier conjunto de polinomios sirve: deben elegirse de modo que se preserve la recuperación local y se controle el grado (para asegurar buena distancia).

La construcción se basa en interpolación de polinomios sobre cuerpos finitos y se puede describir así:

- Se fija un cuerpo finito $\mathbb{F}_q$ y un conjunto $A \subset \mathbb{F}_q$ de n puntos de evaluación. Estos puntos se agrupan en subconjuntos disjuntos $A = A_1 \cup A_2 \cup \cdots \cup A_{n/r}$, cada uno de tamaño $r+1$. Cada subconjunto funcionará como un *grupo de recuperación local*.
- Se elige un polinomio $g \in \mathbb{F}_q[x]$ que sea **constante en cada** A_i. Es decir, g toma un único valor en cada subconjunto A_i, pero valores distintos entre los subconjuntos. Este polinomio permite "colapsar" cada conjunto a un punto distinto del codominio.
- Se define un subespacio $\mathcal{L}$ de polinomios de la forma:

$$\mathcal{L} = \left\{ \sum_{j=0}^{t-1} f_j(x) \cdot g(x)^j \;\middle|\; \deg(f_j) < r \right\}.$$

  Cada término $f_j(x) \cdot g(x)^j$ tiene una componente "local"(el f_j) que vive en un espacio de polinomios de grado menor que r, y una componente "global"que varía entre los subconjuntos.
- Finalmente, se define el código como el conjunto de todas las evaluaciones de polinomios de $\mathcal{L}$ en los puntos de A. Es decir:

$$C = \{(f(\alpha_1), \ldots, f(\alpha_n)) \mid f \in \mathcal{L}\}.$$

La recuperación local se garantiza porque, al restringir $f \in \mathcal{L}$ a un conjunto A_i, se obtiene un polinomio de grado menor que $r+1$, por lo que se puede recuperar cualquier símbolo del grupo mediante interpolación polinómica (por ejemplo, usando el algoritmo de Lagrange).

Esta construcción permite controlar simultáneamente la *dimensión*, la *localidad* y la *distancia* del código. Además, alcanza la cota de Singleton para LRC en muchos casos, lo que la convierte en óptima. A pesar de lo técnico del procedimiento, el resultado es sorprendentemente eficiente y elegante.

Un ejemplo concreto

Veamos cómo funciona todo esto con un caso sencillo en el cuerpo finito con 13 elementos, $\mathbb{F}_{13}$. Tomemos el conjunto

$$A = \{1, 2, 3, 4, 5, 6, 9, 10, 12\} \subset \mathbb{F}_{13},$$

y dividámoslo en tres partes:

$$A_1 = \{1, 3, 9\}, \quad A_2 = \{2, 5, 6\}, \quad A_3 = \{4, 10, 12\}.$$

Cada subconjunto tiene tres elementos ($r + 1 = 3$ con $r = 2$), y serán nuestros grupos de recuperación local.

Ahora necesitamos un polinomio $g(x)$ que sea constante dentro de cada subconjunto. Un candidato simple es

$$g(x) = x^3.$$

En efecto, $g(x)$ vale 1 en todos los elementos de A_1, vale 8 en A_2 y vale 12 en A_3.

A partir de este polinomio construimos el espacio

$$\mathcal{L} = \{\, f(x) = (a + c\, g(x)) + x(b + d\, g(x)) \mid a, b, c, d \in \mathbb{F}_{13} \,\}.$$

En otras palabras, $\mathcal{L}$ está formado por polinomios que combinan una parte "local" (lineal en x) con una parte "global" controlada por $g(x)$, lo que garantiza que al restringirlos a cada conjunto A_i se comporten como polinomios de grado bajo.

Si expandimos, se ve que los polinomios de $\mathcal{L}$ tienen la forma

$$f(x) = a + bx + cx^3 + dx^4,$$

con grado a lo sumo 4.

El código se define evaluando todos estos polinomios en los puntos de A:

$$C = \left\{ (f(1), f(2), f(3), f(4), f(5), f(6), f(9), f(10), f(12)) \;\middle|\; f \in \mathcal{L} \right\}.$$

De esta manera construimos un código de longitud $n = 9$, dimensión $k = 4$, y con recuperación local $r = 2$. Dentro de cada grupo A_i, si falta un símbolo, los otros dos alcanzan para reconstruirlo usando interpolación lineal. Además, como los polinomios tienen grado a lo sumo 4, la distancia mínima es al menos 5.

Veamos la recuperación en acción. Supongamos que el mensaje es

$$m = (1, 1, 1, 1),$$

que corresponde al polinomio

$$f(x) = 1 + x + x^3 + x^4.$$

La palabra código asociada es

$$c_m = (4, 8, 7, 1, 2, 11, 0, 0, 0).$$

Imaginemos que durante la transmisión se pierde el primer símbolo. Como corresponde a $f(1)$ y $1 \in A_1$, podemos usar los otros dos valores de ese grupo: $f(3) = 8$ y $f(9) = 7$. Con ellos construimos el polinomio interpolador de grado 1, que resulta ser $\delta(x) = 2x + 2$. Al evaluarlo en 1, obtenemos $\delta(1) = 4$, recuperando así el símbolo perdido.

Este ejemplo pequeño muestra con números concretos cómo funciona la idea: la construcción de $\mathcal{L}$ usando $g(x)$ asegura que cada grupo A_i permite una recuperación local eficiente, a la vez que el código conserva buena distancia mínima.

Y aún hay más...

El estudio de códigos con recuperación local está lleno de variantes y todavía hay muchas preguntas abiertas. Algunas investigaciones se proponen buscar códigos que toleren múltiples fallas simultáneas en el mismo grupo de recuperación, otros exploran versiones adaptadas a diferentes modelos de red o a nuevas arquitecturas de memoria.

Además, hay conexiones inesperadas con otras áreas: combinatoria, grafos, teoría de matroides y hasta geometría algebraica (con evaluaciones de puntos racionales en curvas definidas sobre cuerpos finitos). En resumen, es un ejemplo perfecto de cómo una necesidad tecnológica concreta puede impulsar desarrollos teóricos profundos.

Conclusión

Los códigos con recuperación local son una respuesta matemática elegante a una necesidad tecnológica muy concreta. Combinan ideas del álgebra, la combinatoria y la teoría de la información para diseñar estructuras que protegen nuestros datos de forma eficiente y accesible.

Lo que comenzó como un problema abstracto sobre polinomios y cuerpos finitos hoy permite que nuestras fotos, documentos y mensajes estén seguros incluso si un servidor se apaga, una red se cae o un disco se rompe. Y todo eso, gracias a un poco de matemática abstracta.

Referencias

[1] Tamo, I., & Barg, A. (2014). A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 60(8), 4661–4676.

[2] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, *Erasure coding in Windows Azure Storage*, USENIX Annual Technical Conference (USENIX ATC), 2012. Disponible en: https://www.microsoft.com

Ir al índice general