

COMPRESSSIONE

L'operazione di compressione è un processo di trasformazione dei dati in un'altra rappresentazione. Perché è utile comprimere?

- 1) Salvare spazio nello storage
- 2) Salvare tempo di trasmissione dei dati

È importante che questo processo sia **reversibile**, preferibilmente in modo esatto, ma possono essere tollerate delle perdite.

Trade-off fra quanto spazio si risparmia e quanto tempo si spende per eseguire le operazioni sui dati compressi (Vanno prima decompressi)

Modello



Da qui definiamo il rapporto di compressione (compression ratio) CR:

$$CR = \frac{B}{C(B)}$$

Che indica di quale fattore l'output è più piccolo del l'input.

Nessun algoritmo può comprimere ogni bit-string.

Questa affermazione si può facilmente provare per assurdo, assumendo che:

$$B > C(B) > C(C(B)) > C(C(C(B))) \dots$$

ovvero che, se comprimiamo in loop arriviamo ad avere ∞ bit: ASSURDO!

ZIPF'S LAW

La legge di Zipf tenta di stimare le frequenze relative dei termini del vocabolario.

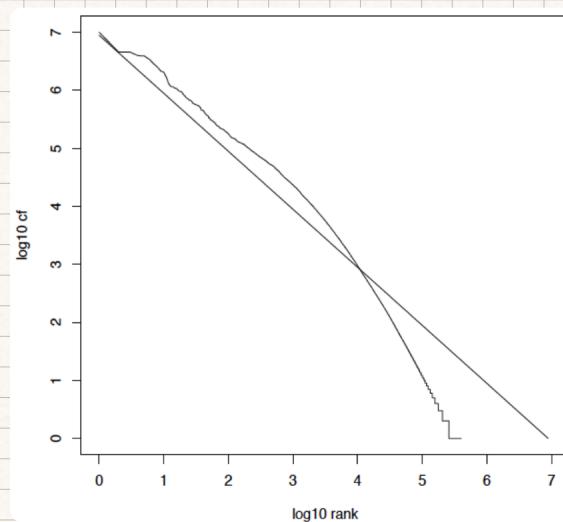
Se ordiniamo i termini per frequenza, l' i -esimo termine avrà frequenza proporzionale a $1/i$. Quindi considerando la collection-frequency del termine i -esimo c_{fi} :

$$c_{fi} \propto \frac{1}{i} = \frac{K}{i}$$

Dove K è una costante per normalizzazione.

Nello spazio Logaritmico:

$$\log c_{fi} = \log K - \log i$$



CONSEQUENZE

La conseguenza delle due leggi di HEAP e ZIPP è che la dimensione del vocabolario non ha un limite superiore, per tanto continuerà a crescere insieme alla collezione di documenti.

Ma lo spazio più grosso è occupato dalle posting lists.

Vanno comprimate!

COMPRESSSIONE DATI INTEGRI

Abbiamo un intero x e lo vogliamo rappresentare con meno bit possibili. Assumiamo $x > 0$.

La rappresentazione di x secondo il **codice** usato per comprenderlo (l'algoritmo) è detta **Codeword** $c(x)$.

I codeword che vedremo sono **statici** perché dato x eseguiamo sempre lo stesso $c(x)$.

BINARY CODE

Indichiamo con $\text{bin}(x, K)$ la rappresentazione binaria di x usando K bits: $0 \leq x \leq 2^K - 1$.

Se invece scriviamo solamente $\text{bin}(x)$, assumiamo che $K = \lceil \log_2(x+1) \rceil$, ovvero il numero minimo di bit per rappresentare x .

Esempio: $x = 16$, $K = \lceil \log_2(17) \rceil = 5$, $\text{bin}(16) = 10000$.

Dato che consideriamo che $x > 0$, possiamo dire che $B(x) = \text{bin}(x-1) = \text{binary Codeword}$.

La dimensione di ogni $c(x)$ per binary è:

$$c(x) \geq \lceil \log_2(x) \rceil = \text{bin}(x-1) = B(x)$$

$B(x)$ è un limite inferiore di $c(x)$.

Binary Code:

x	$B(x)$
1	0
2	1
3	10
4	11
5	100
6	101
7	110
8	111

Comprimiamo i binary codeword.

Dato che $C(x) > B(x)$ data una sequenza di numeri $L = [x_1 \dots x_n]$ possiamo codificare L come $B(x_1) \dots B(x_n)$.

Esempio: $L = [3, 5, 2] = 101001$

Ma così facendo non possiamo tornare indietro, troppi modi per leggere questa stringa di bit (Possiamo per esempio leggere 6, 1, 1, 2). Problema dell'ambiguità.

PREFISSO

Bisogna aggiungere un prefisso per ogni codeword, così da capire quando finisce uno e ne inizia un altro.

Una codifica C è detta **prefix-free** quando non ci sono $C(x)$ e $C(y)$: $C(y) \geq C(x)$ per cui

$$C(x) = C(y) \cdot [\emptyset : C(x)-1]$$

In parole povere nessuna codifica ha una sequenza di bit che è uguale al prefisso di un'altra codifica. Quindi è senza ambiguità.

BINARY

x	$B(x)$
1	0
2	1
3	10
4	11
5	100
6	101
7	110
8	111

PREFIX-FIX

x	$C(x)$
1	00
2	01
3	100
4	101
5	1100
6	1101
7	11100
8	11101

UNARY

Un $x > \emptyset$ viene rappresentato come $U(x) = 1^{x-1} \emptyset$, ovvero come una sequenza di $(x-1)$ unità e uno \emptyset finale.

x	$U(x)$
1	0
2	10
3	110
4	1110
5	11110
6	111110
7	1111110
8	11111110

Questa codifica è buona solo per interi molto piccoli!

GOMMA

Scriviamo $b = bin(x)$ uscendo unary $U(b)$. Il che vuol dire prendere la rappresentazione binaria di x , vedere se quanti bit sta, diciamo K , e fare l'unoario di K . Però così abbiamo le collisioni, ad esempio i numeri 2 e 3 hanno entrambi rappresentazione se $K=2$ bit, $U(2) = 10$. Per questo si fa un seguire le $(K-1)$ cifre meno significative della rappresentazione binaria di x : $2 \Rightarrow 10.0$ | $3 \Rightarrow 10.1$

DELTA

Rimpiazza la parte $U(b)$ di γ con $\delta(b)$. Questo perché $U(b)$ è molto grande per grandi numeri. La seconda parte rimane uguale.

x	$\gamma(x)$	$\delta(x)$
1	0.	0.
2	10.0	100.0
3	10.1	100.1
4	110.00	101.00
5	110.01	101.01
6	110.10	101.10
7	110.11	101.11
8	1110.000	11000.000

VARIABLE BYTES

L'idea è comprimere byte per byte invece che bit per bit. Così che, allineando al byte, diventa più conforme a come i calcolatori allineano in memoria.

In particolare, variable byte scrive l'intero $x > 0$ su i vari byte necessari dividendo, per ogni byte i bit presi dalla rappresentazione binaria di x , e l'8° bit detto **control bit**, è 1 se il prossimo byte è la continuazione del numero attuale compresso e 0 altrimenti.

Esempio:

$$x = 67822 \quad \text{bin}(x, k=17) = \underline{10000100011101110}$$

00000100 10010001 11101110

Il fatto che variable byte lavora allineando al byte favorisce la **semplicità dell'implementazione** e la **velocità di decompressione** a discapito della efficienza della compressione stessa.

INFORMATION CONTENT

L'idea alla base è che un evento raro ha con sé un information content maggiore, per la complessione se un intero x ha una probabilità di apparire $P(x)$ bassa, allora possiamo permetterci di rappresentarlo su più bit rispetto a un intero che ha $P(x)$ più grande. Per esempio per le frequenze meglio usare unary poiché sono formate da numeri per lo più piccoli.

Definizione Information Content

$$I(x) = \log_2 \left(\frac{1}{P(x)} \right) = -\log_2 P(x)$$

Ragionando in binario usiamo \log_2 .

Quindi più è grande $P(x)$ più è piccolo $I(x)$.

ENTROPIA

Presso l'intero x con $I(x)$ definiamo l'entropia $H(P)$:

$$H(P) = \sum_x P(x) I(x) = -\sum_x P(x) \log_2 P(x)$$

È facile provare che $H(P) \geq 0$.

L'entropia $H(P)$ rappresenta la lunghezza ottimale di un codeword. Difatti è il limite inferiore di bit necessari a codificare x con $C(x)$.

CODIFICA OTTIMA (optimal code)

Per essere ottimo un codeword sarà:

$$|C(x)| = I(x) = -\log_2 P(x)$$

Da cui ricaviamo che $P(x) = 2^{-|C(x)|}$

Esempi:

- Unary: $P(x) = 1/2^x$
- Binary: $P(x) = 1/U$, if each x is less than U and coded in $\lceil \log_2 U \rceil$ bits
- Gamma: $P(x) \approx 1/(2x^2)$
- Delta: $P(x) \approx 1/(2x(\log_2 x)^2)$
- Variable-Byte: $P(x) \approx \sqrt[7]{1/x^8}$

COMPRESIONE DELLE LISTE DI INTERI

A volte conviene comprimere le liste e non gli interi singolarmente. Possiamo farlo assumendo che la lista da comprimere sia **ordinata**.

Assumiamo che la lista sia fatta di interi **distinti**, messi in ordine **crescente**.

Questi metodi non contemplano l'aggiornamento della lista.

Ci sono $\binom{U}{n}$ combinazioni fatte da n numeri interi distinti in un universo $U \geq n$. Per questo usiamo **L'APPROSSIMAZIONE DI STIRLING**:

$$\log_2(n!) \approx n \log_2 n - n \log_2 e$$

Abbiamo bisogno di $\lceil \log_2 \binom{U}{n} \rceil$ bits, per rappresentare una lista fatta da n interi:

$$\lceil \log_2 \binom{U}{n} \rceil \approx n \left(\log_2 \left(\frac{U}{n} \right) - \log_2 e \right) + 1.44n \text{ bits}$$

Questa affermazione è vera solo quando assumiamo che ogni tipo di lista sia equiprobabile. Ma nel mondo reale i dati non sono completamente casuali!

GAPS

Usa la proprietà che la differenza fra due interi consecutivi è, in media, più piccola degli interi della lista originale:

$$\bullet L = [1, 3, 13, 14, 15, 16, 20, 22, 23, 34, 35, 36, 40]$$

$$\bullet L' = [1, 2, 10, 1, 1, 1, 4, 2, 1, 11, 1, 1, 4]$$

Comprimiamo poi ogni intero della lista dei gaps con un codice per singoli interi.

next G&Q access: per usarla è necessario decomprimere prima.

In letteratura è usata per comprimere i docId.

BLOCKING

L'idea è quella di dividere la lista in blocchi e codificare ogni blocco a se. I blocchi possono essere sia di lunghezza variabile che fissa.

Questa strategia è molto utile se per esempio i gaps all'interno di un blocco sono molto piccoli.

Esempio:

$$L = [1, 3, 13, 14, 15 | 16, 20, 22, 23, 34 | 35, 36, 40, 48, 51 | 52, 53, 54]$$

$upper = [15, 34, 51, 54]$ (encoding of the last elements of the blocks)

$lower = [1, 2, 10, 1] [1, 4, 2, 1] [1, 1, 4, 8] [1, 1]$ (gaps of the integers in the blocks)

$widths = [4, 3, 4, 1]$ (all gaps from block i can be represented in $widths[i]$ bits)

→ tutti i gaps del primo blocco stanno al massimo su 4 bit ($10_b \rightarrow 1010$)

nextG&Q(x): Si può cercare x usando la ricerca binaria sui blocchi, alla ricerca del blocco che ha $\max \text{Doc Id} < x$. Poi decomprimere solo quel blocco.

- Per un blocco con B interi è $O(\log(n/B) + B)$

access(i): l' i -esimo intero sta nel blocco $[i/B]$

- Complessità $O(B)$

P-FOR-DELTA

Supponiamo di avere la seguente lista

$$[1, 1, 1, 1, 1, 1, 1, 8247, 1, 1, 1, 1]$$

Dove solo il numero 8247 fa sì che la rappresentazione binaria sia troppo grande per gli altri numeri. In particolare si usano $\lceil \log_2 8247 \rceil = 14$ bits, ma solo il numero più grande necessita davvero di 14 bits.

L'idea di PForDelta è quella di scegliere una base b e un valore $K > 0$ tale per cui la maggior parte dei numeri della lista ($\approx 80\%$) cade nell'intervallo numerico $[b, b+2^K-1]$.

Gli interi che cadono in questo intervallo vengono scritti come il delta $x-b$ usando K bits.

Ogni intero che non sta nell'intervallo e che quindi $x > b+2^K$ viene assegnato il codeword speciale 2^K-1 e codificato poi in una lista separata.

Esempio

$$L = [3, 4, 7, 21, 9, 12, 5, 16, 6, 2, 34] \text{ con } b=2 \text{ e } K=4.$$

L'intervallo è $[2, 15]$

$$[1, 2, 5, *, 7, 10, 3, *, 4, 0, *] - [21, 16, 34]$$

Codifichiamo i gap come:

0001.0010.0101.1111.0111.1010.0011.1111.0100.0000.1111

SIMPLUS 9

L'idea è che, in 32 bits posso contenere molti numeri interi. Bisogna però codificarli in qualche modo.

Simple 9 divide i 32 bits in:

1) 4 bit più significativi, a indicare come sono divisi i numeri nella lista, nei restanti 28 bit. Se nei 4 bits più significativi c'è scritto:

- 0: c'è un solo numero su 28 bits
- 1: due numeri su 14bit
- 2: tre numeri su 9bit, 1bit spesso
- 3: quattro numeri su 7bits
- 4: cinque numeri su 5bits, 3bits sparsi
- 5: sette numeri su 4 bits
- 6: nove numeri su 3 bits, 1bit spesso
- 7: quattordici numeri su 2 bits
- 8: ventotto numeri su 1 bit

Glias-Fano

Prese una lista di interi, la rappresentazione binaria di ogni intero in $\lceil \log_2 U \rceil$ bits (Dove U è l'intero più grande) viene divisa in due parti:

1) **Low Bits**, i bit meno significativi, che sono in numero $\ell = \lceil \log_2 (U/n) \rceil$

2) **High Bits**, i bit più significativi, che sono in numero $\lceil \log_2 U \rceil - \ell$

I low-bits vengono scritti in un array così come sono in sequenza in un array grande $n\ell$ bits.

Mentre gli high-bits sono clusterizzati insieme e codificati in unary, codificando il numero di interi presente nel cluster: Se ci sono tre elementi avremo $U(3+1) = 1110$. Il tutto scritto in un altro vettore binario.

- The **binary representation** of each integer x in $\lceil \log_2 U \rceil$ bits is split into **two parts**: the least most significant $\ell = \lceil \log_2 (U/n) \rceil$ bits and the remaining $(\lceil \log_2 U \rceil - \ell)$ bits — the **low** and **high bits**, respectively.
- The **low bits** are **written in a vector**, `low_bits`, of $n\ell$ bits (each integer takes ℓ bits)
- The **high bits** are **clustered together** and written in **unary** in another bit-vector `high_bits`

$$\begin{aligned} \text{high_bits} &= 1110.1110.10.10.110.0.10.10 \\ \text{low_bits} &= 011.100.111.101.110.111.101.001.100.110.110 \end{aligned}$$

L	$\lceil \log_2 U \rceil$
3	000.011
4	000.100
7	000.111
13	001.101
14	001.110
15	001.111
21	010.101
25	011.001
36	100.100
38	100.110
54	110.110
62	111.110

Il costo dei low-bits è $n \cdot l$.

Il costo degli high-bits è $2n$ al massimo.

I cluster sono in numero $U/2^l + 1$ (Al massimo n)

Lo spazio che può raggiungere al massimo Elias-Fano è

$$n \cdot \lceil \log_2 \frac{U}{n} \rceil + 2n$$