Teoria dell'Informazione e Crittografia

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.

Capitolo 2

Introduzione agli argomenti del corso

Si ha una sorgente che emette messaggi e li vuole mandare tramite un canale di comunicazione, che contiene del rumore che agisce sui messaggi e li rovina. Il destinatario per capire che il messaggio è rovinato ha varie tecniche. Una prima cosa che potrebbe fare è chiedere di rimandare il messaggio ma sarebbe meglio correggere *in loco* e si hanno algoritmi per farlo (tra cui lo **schema di Hamming** usando il modello del **rumore bianco**).
Un'altra tematica è la codifica stessa del sorgente, comprimendo flussi di dati, senza perdere informazioni.
Si parlerà anche dei canali di comunicazione, delle capacità e dei **teoremi di Shannon**.
Si vedranno poi le basi della crittografia, i crittosistemi storici, vari standard etc...

Capitolo 3

Teoria dell'informazione

Si hanno due sottoparti principali:

- **teoria dei codici**, per individuare e correggere gli errori. Si studia il canale di trasmissione cercando di contrastare il rumore che c'è nel canale
- **teoria dell'informazione**, in cui il focus è la sorgente delle informazioni

Vediamo il classico schemino della teoria dell'informazione:

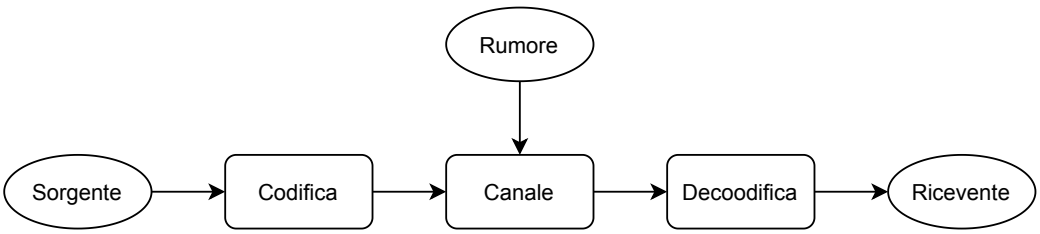


Figura 3.1: Schema di un sistema generale di comunicazione tipico della teoria dell'informazione

Avendo:

- la **sorgente** che produce segnali, dei simboli, che potrebbero essere continui (come la corrente), anche se noi li assumeremo come simboli di un alfabeto finito, avendo quindi una **sorgente discreta**
- i simboli, per poter essere spediti all'interno di un canale, vanno codificati, avendo una parte di **codifica**

- una volta codificati i simboli vanno nel **canale di trasmissione**, dove si ha del **rumore**. Tale *rumore* prende un simbolo di quelli inseriti e lo cambia
- dal canale esce o il simbolo che è entrato o il simbolo modificato dal rumore e, tipicamente, non è immediatamente utilizzabile ma deve passare per una fase di **decodifica**
- il simbolo decodificato arriva al **destinatario**

Si hanno alcune assunzioni sulla sorgente:

- è **discreta**, i simboli emessi appartengono ad un alfabeto finito. Normalmente tali simboli sono $S = \{s_1, s_2, \ldots, s_q\}$
- i simboli vengono emessi uno alla volta ad ogni **colpo di clock**. Non si ha mai che in un colpo di clock non escano simboli o che ne escano più di uno solo
- la sorgente è **senza memoria (*memoryless*)**, avendo che i simboli già usciti non influenzano per nulla il simbolo che sta per uscire. Ogni simbolo che esce non tiene conto del passato, *è come se fosse il primo*
- è **probabilistica e randomizzata**. Si ha quindi che i simboli $S = \{s_1, s_2, \ldots, s_q\}$ escono con le probabilità $(p_1, p_2, \ldots, p_q)$. Deve valere che, ovviamente, che $p_i \in [0, 1], \forall i = 1, \ldots, q$. Si ha inoltre che le varie probabilità, nel loro insieme, devono formare una distribuzione di probabilità, avendo che:

  $$\sum_{i=1}^{q} p_i = 1$$

  Potrei avere simboli con probabilità nulla di comparire ma nella pratica non è qualcosa di sensato. La sorgente la costruisco o da zero (e a quel punto un simbolo con probabilità nulla non lo metterei) o ho una sorgente che devo studiare (e qui potrei avere simboli con probabilità bassissime se non nulle, in tal caso bisognerebbe rivalutare l'assunzione dei simboli di quella sorgente). Possiamo quindi meglio dire che $p_i \in (0, 1], \forall i = 1, \ldots, q$.
  D'altro canto vedo se posso avere probabilità pari a 1 per un simbolo ma in tal caso avrei solo quello e non sarebbe interessante. Si ha quindi che:

  $$p_i \in (0, 1), \forall i = 1, \ldots, q$$

$$\sum_{i=1}^{q} p_i = 1$$

Le sorgenti che emettono i simboli secondo uno schema prefissato, deterministico, sono poco interessanti, avendo un comportamento banale

Il concetto di *spedire in un canale* può anche essere generalizzato in altre "idee", come il disco su cui salvo dei dati e il tempo per cui li salvo.
La parte di *codifica e decodifica* può essere approfondita. Nello schema in figura 3.1 ci si è infatti concentrati sul canale, avendo che la codifica serve a fare in modo che il simbolo trasmesso vada bene per essere trasmesso nel canale. La **codifica** è a sua volta suddivisa in due parti:

1. **codifica di sorgente**, che ha come obiettivo rappresentare nel modo più efficiente e compatto i simboli emessi dalla *sorgente*. Si vuole quindi comprimere la sequenza di simboli (messaggi) emessi dalla sorgente, per impegnare meno banda possibile quando andremo a spedire. Si deve considerare che ogni bit in un file compresso è essenziale per permettere di poter recuperare il contenuto compresso

2. **codifica di canale**, che ha quasi uno scopo opposto rispetto alla *codifica di sorgente*, infatti ha come obiettivo quello di contrastare il rumore e per farlo aggiunge ridondanza al messaggio (da qui il discorso sull'obiettivo opposto)

Si cerca quindi di comprimere il più possibile nella prima fase, quella di *codifica di sorgente* e di ridondare il meno possibile nella seconda, quella di *codifica di segnale*.
Per capire meglio quanto detto diamo alcune formalità.

Definizione 1. *Una **codifica** è una funzione cod che prende i simboli della sorgente $S = \{s_1, s_2, \ldots, s_q\}$ e ad ogni simbolo s_i gli assegna una stringa formata coi caratteri di un certo alfabeto Γ, l'**alfabeto della codifica**. Le stringhe di Γ^* sono tutte quelle costruite sull'alfabeto Γ di lunghezza arbitraria e finita, compresa la stringa vuota ε, che posso quindi formare coi simboli di Γ. Quindi ad ogni $s_i \in S$ assegno un $\gamma_i \in Gamma^*$, avendo che:*

$$cod : S \to \Gamma^*$$

generalmente si ha che:

$$|\Gamma| < |\Sigma|$$

e quindi i simboli di Σ *sono mappati da cod in sequenze di simboli di* Γ*, a meno che non si ritenga accettabile il fatto che due o più simboli di* Σ *vengano mappati nellostesso simbolo di* Γ*.*
Più avanti nel corso vedremo casi in cui $|\Gamma| > |\Sigma|$

Si hanno quindi i simboli $S = \{s_1, s_2, \ldots, s_q\}$ che escono con probabilità $(p_1, p_2, \ldots, p_q)$ e che vengono codificati con le stringhe $\gamma_1, \gamma_2, \ldots, \gamma_q$. Le varie $gamma_i$ sono dette **codeword**. Chiamando $l_i = |\gamma_i|$ la lunghezza di tali stringhe si ha che tali stringhe hanno associati i vari $l_1, l_2, \ldots, l_q$.
L'obiettivo quindi della *codifica di sorgente* è quello di minimizzare la lunghezza media L delle stringhe, avendo quindi una media pesata (pesata sulle probabilità):

$$L = \sum_{i=1}^{q} p_i \cdot l_i$$

Tenendo conto delle probabilità, per minimizzare L, si deve, avendo a che fare con termini che sono tutti > 0 (avendo supposto che non si hanno probabilità nulla e avendo che una codeword pari alla stringa vuota ha poco senso), fare in modo che i termini siano tutti il più piccolo possibile. Dato che le probabilità sono date mentre la codifica la sto costruendo, calcolando le codeword e di conseguenza le loro lunghezze, devo fare in modo che se la probabilità è grande la lunghezza deve essere piccola. Se invece la probabilità è piccola posso permettermi una lunghezza più grande. Parto quindi dai simboli con probabilità più grande e inizio a usare codeword più piccole possibili, usando via via quelle più lunghe.
Un'idea simile è usata nel *codice Morse* dove le lettere meno comuni hanno le sequenze più lunghe di punti, linee e spazi (avendo una codifica ternaria). La lettera più comune, la "e", ha infatti solo con un punto, la codifica più breve mentre le meno comuni hanno sequenze multiple di punti, linee e spazi che le separano (e gli spazi contano nella lunghezza di queste codeword).
Noi non sappiamo in anticipo che messaggi verranno prodotti dalla sorgente e quindi le codeword vanno studiate passo a passo, valutando i simboli più probabili per associare le codeword più brevi e i meno probabili per le codeword più lunghe.
Analizziamo meglio i codici, le codeword. Possono essere:

1. *a lunghezza fissa*, ovvero si ha che $l_1 = l_2 = \cdots = L_q$
2. *a lunghezza variabile*, avendo che ogni codeword può avere lunghezza diversa

Ne segue quindi che il discorso di minimizzare L ha senso solo in presenza di *codeword a lunghezza variabile* (potendo decidere per ogni simbolo che

codeword associare), avendo la **codifica a lunghezza variabile**.
Con *codeword a lunghezza fissa* avrei tutte le l_i uguali e quindi avrei, avendo $l_i = l, \forall i$:

$$L = \sum_{i=1}^{q} p_i \cdot l_i \sum_{i=1}^{q} p_i \cdot l = l \cdot \sum_{i=1}^{q} p_i = l \cdot 1 = l$$

avendo, come facilmente intuibile, che la lunghezza media è la lunghezza fissa stessa. Si hanno codifiche a lunghezza fissa, come banalmente numeri a 64bit etc... in tal caso si parla di **codici a blocchi**.
Usando codifiche a lunghezza fissa si hanno anche esempi interessanti come quello del *codice pesato*, detto **codice pesato 01247**. Il nome deriva dal fatto che si possono codificare le cifre da 0 a 9 (da 1 a 9 con poi lo 0 dopo il 9) sotto forma di stringhe di 5 bit usando i pesi 0,1,2,4,7 associati a ciascun bit. Vediamo la tabella con la codifica di questo codice:

	0	1	2	4	7
1	1	1	0	0	0
2	1	0	1	0	0
3	0	1	1	0	0
4	1	0	0	1	0
5	0	1	0	1	0
6	0	0	1	1	0
7	1	0	0	0	1
8	0	1	0	0	1
9	0	0	1	0	1
0	0	0	0	1	1

Si nota che ogni codeword ha sempre 3 bit pari a 0 e due bit pari a 1, avendo un **codice 2-su-5**. Banalmente i pesi si associano ai numeri 0,1,2,4,7 in modo tale che essi, sommati, formino il numero voluto (ad esempio per 1 avrò i pesi su 0 e 1, per 9 su 2 e 7 etc...). L'unico caso è il caso dello 0, che non può essere ottenuto come somma di due pesi (spesso si hanno nei codici casi speciali da gestire a parte). Per lo 0 viene quindi presa una codeword non usata per altri numeri e quindi l'unica scelta possibile è avere i pesi su 4 e 7 (visto che farebbe 11).
Su un totale ci 5 bit, avendo due bit a 1 e tre bit a 0, posso avere un numero di codeword pari a:

$$n = \binom{5}{2} = 10$$

avendo che i 5 bit sono associati ai 5 elementi dove 1 segnala che "sto usando quel peso", prendendo quindi i sottoinsiemi di due elementi a partire da un

insieme di cinque elementi, ovvero "in quanti modi posso formare sottoinsiemi che contengono due elementi a partire da un insieme di cinque elementi" o detto altrimenti "quanti sono i modi in cui posso disporre due uni all'interno di una stringa di cardinalità cinque".
In un linguaggio di programmazione privo di una struttura dati dedicata posso simulare un insieme di questo tipo tramite un vettore di bit (con 1 se l'elemento associato all'indice c'è).
Il codice a barre è detto ***codice 39*** *ed è un* ***codice 3-su-9***.
In merito alla **decodifica** si ha che anch'essa sarà di due tipi:

1. **decodifica di canale**, vedendo e c'è stato un errore di trasmissione ed eventualmente correggendolo in automatico se il codice mi consente di farlo

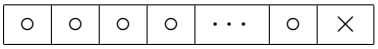
2. **ulteriore decodifica** che non è esattamente una *codifica di sorgente* ma quanto una *trasformazione*, dove le *codeword* vengono trasformate nel formato leggibile dal **ricevente**

3.1 Codici per individuare errori

Ci concentriamo ora sulla *codifica di canale* ignorando per ora la *codifica di sorgente*, avendo come obiettivo l'aggiunta di ridondanza a simboli, che si suppongono già codificati con codeword, in modo tale che in queste codeword, spedite nel canale dove eventualmente si possono avere modifiche causate dal rumore, vengano eventualmente riconosciuti (ed eventualmente corretti) errori in fase di *decodifica*.
Parlando di codici per individuare errori solitamente, nei disorsi, si ha che $|\Gamma| < |\Sigma|$ Qualora il ricevente con la sua decodifica si accorga che è successo qualcosa ma non si è in grado di correggere quel qualcosa si hanno i cosiddetti **codici per individuare gli errori (*error detection codes*)**. Nel caso in cui il ricevente con la sua decodifica si accorga dell'errore ci si chiede anche se può correggerlo autonomamente senza chiedere che la sorgente spedisca nuovamente il messaggio. Non sempre questa cosa si può fare ma quanto accade si parla di **codici a correzione d'errore (*error correction codes*)**.
Vediamo come capire se un messaggio ricevuto è valido.
Si supponga di spedire un pacchetto di n bit (ma potrebbe essere qualsiasi altra cosa ma per praticità prendiamo un bit) nel canale e che da esso esca un certo pacchetto sempre di n bit (per il rumore potrebbe non essere lo stesso).

Definizione 2. *Definiamo il* ***controllo di parità****.*
Avendo una sequenza di n bits in cui si ha $n-1$ bits, dette ***cifre di messaggio*** *di* ***messaggio vero****, che chiameremo* ***msg*** *e un bit che è la* ***cifra***

***di controllo**, che chiameremo **check**. Le cifre di messaggio si indicano con* $\circ$ *mentre la cifra di controllo con* $\times$ *e quindi il messaggio è del tipo:*



avendo $n-1$ $\circ$ *e un solo* $\times$ *(che potrebbe anche non essere in fondo, basta avere coscienza della posizione nel pacchetto, concordando la cosa tra mittente e ricevente).*
Si ha che:

- *chi spedisce ha le cifre di messaggio e deve calcolare la cifra di controllo*
- *chi riceve controlla che la cifra di controllo sia coerente con le cifre di messaggio*

*Nell'**error detection code** il ricevente è solo in grado di capire che la sequenza non è valida ma per farlo bisogna assumere di avere limitazioni nella sequenza di* n *bit che è entrata nel canale. Questa limitazione è che gli* n *bit entranti nel canale siano una **codeword valida**, avendo che, preso un sottoinsieme* M *di tutto l'insieme di* n *bit, ovvero* $M \subseteq \{0,1\}^n$*,* M *è un insieme di codeword valide. Quindi solo un messaggio appartenente a* M *può entrare nel canale. Fatta questa premessa, quando esce un messaggio, ho che, a causa del rumore, questo messaggio viene rovinato, non avendo più un messaggio valido (cosa che viene capita dal ricevente). Purtroppo può succedere che il rumore trasformi un messaggio valido in un altro messaggio valido ma non considereremo questa opzione per ora.*
*Nel **controllo di parità semplice** il pacchetto di* n *bit, come visto è formato da* $n-1$ *bit di messaggio e un bit di controllo, la **check digit**. Si procede quindi, ricordando che siamo in un caso binario, a contare il numero di* 1 *nei primi* $n-1$ *bit e se questo è dispari setto il bit di check a* 1 *e si nota che così il numero di* 1 *nel pacchetto intero di* n *bit diventa pari, avendo la cosiddetta **parità pari** (ovvero ogni sequenza valida ha un numero pari di* 1*). Controllando il numero di* 1 *il ricevente capisce se il rumore ha modificato il messaggio anche se non può capire cosa è successo (avendo quindi che il* controllo di parità semplice *è solo un* error detecting code *e non un* error correcting code*). Non potendo fare nulla, in caso di errore identificato, il ricevente può solo chiedere al mittente di inviare nuovamente il messaggio. Qualora il rumore modificasse il messaggio in modo tale che si abbia comunque un numero apri di* 1 *si rientrerebbe nella casistica sopra descritta in cui il rumore forma ancora un messaggio valido. Questa cosa può succedere se, nel caso binario, il rumore modifica un numero pari di cifre e quindi il* controllo

di parità semplice *funziona solo se viene modificato un* numero dispari di cifre.

Vediamo quindi meglio come calcolare la **check digit**.
Si rinominiamo gli n bit come:

$$x_1 x_2 \cdots x_{n-1}\, y$$

quindi con y *check digit*.
Si ha che, con $\oplus$ *xor*:

$$y = x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1}$$

Ricordando che:

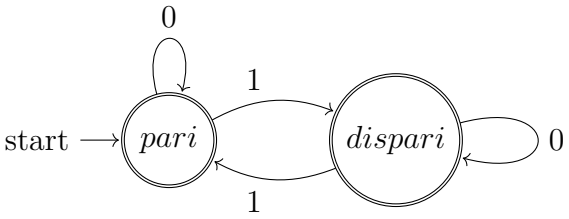
a	b	$a \wedge b$	$a \oplus b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

quindi vale 1 sse i due bit in input sono diversi ma questo non ci aiuta su $n-1$ input. Altrimenti si ha che vale 1 sse il numero di 1 in input è dispari e questo ci aiuta su $n-1$ input infatti la generalizzazione dello *xor* a più di due input è detta **funzione di parità**. Un altro punto di vista per considerare lo *xor* è quello della **somma a modulo 2** usando la notazione:

$$y = \bigoplus_{i=1}^{n-1} x_i = \sum_{i=1}^{n-1} x_i \bmod 2$$

avendo che faccio prima la somma e poi il modulo 2 mi dice 0 se è pari e 1 se è dispari.
Si ha inoltre una relazione interessante tra le formule scritte usando solo $\oplus$ e $\wedge$ nella cosiddetta **forma algebrica normale (*ANF*)**. Queste formule booleane possono essere trasformate in formule aritmetiche con *modulo 2*, quindi in $\mathbb{Z}_2$ dicendo che lo *xor* equivale alla *formula modulo due* e l'*and* al *prodotto modulo due* (la cosa vale in entrambi i versi).
La funzione di parità è così usata che in tutti i microprocessori, fin dagli anni settanta, si ha un *flag di parità* tra i flag della CPU, che viene settato come appena visto a seconda dei bit caricati su un registro particolare (a volte detto *accumulatore*). Tale calcolo è facilmente mappabile in un circuito, avendo che lo *xor* gode della proprietà associativa (avendo un circuito che fa un albero di porte *xor*).

Posso anche simulare lo *xor* con un **automa a stati finiti**, con due stati "pari" e "dispari", con il "pari" stato iniziale (diciamo che input vuoto è pari):



Questo metodo ha senso se il canale è pochissimo rumoroso, avendo pochissima probabilità di avere la modifica di un bit e ancora meno di due (due modifiche si ricorda che non verrebbero rilevate essendo pari), così poca da poter ipotizzare che non avvengano mai due errori (e se mai dovesse succedere bisognerà valutare l'impatto del problema e le conseguenze). Stiamo assumendo quindi che la **probabilità d'errore** può essere **trascurabile** infatti canali di buona qualità dovrebbero sbagliare non più di un bit su un milione, per canali più affidabili anche uno su un miliardo. Possiamo quindi trascurare che possano accadere due errori e dire che il **controllo di parità** va bene.
Parliamo ora meglio di **ridondanza**, definendola formalmente.

Definizione 3. *La* ***ridondanza*** *R è definita come:*

$$R = \frac{\textit{il numero totale di simboli/cifre spediti}}{\textit{numero di simboli/cifre che sono effettivamente parte del messaggio}}$$

Nel caso del controllo di parità *i simboli che vogliamo spedire sono n bit a fronte di $n-1$ bit di vero messaggio. Si ha quindi:*

$$R = \frac{n}{n-1} = \frac{(n-1)+1}{n-1} = 1 + \frac{1}{n-1}$$

Mettendo in evidenza che la ridondanza è sempre $R \geq 1$, visto che a numeratore abbiamo almeno una cifra in più (quella della check digit*) e che quindi è sicuramente maggiore del denominatore. In realtà per avere $R = 1$ dovrei avere numeratore e denominatore uguali che non ha molto senso parlando di ridondanza, quindi nei casi* interessanti *si ha che $R > 1$. Guardando la formula la cosa è confermata da $1 + \frac{1}{n-1}$ con $\frac{1}{n-1}$ che viene detto* ***eccesso di ridondanza****.*

Definizione 4. *Possiamo* ***generalizzare*** *la definizione di* ***ridondanza****:*

$$R = \frac{msg + check}{msg}$$

avendo:

- *msg numero di simboli/cifre di messaggio*
- *check numero di cifre di controllo*

Ma allora (avendo check < msg per avere qualcosa di sensato):

$$R = \frac{msg + check}{msg} = 1 + \frac{check}{msg}$$

che è la forma "generale" della ridondanza. Si ha che $\frac{check}{msg}$ *è* ***eccesso di ridondanza****.*

Si può dire di non avere necessità di "proteggere" di più il bit di parità in quanto, per la macchina, conta come tutti gli altri. Tutti vanno "protetti" nello stesso modo.
Introduciamo ora il **modello del rumore bianco**