Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Matematica

ASPETTI MATEMATICI DI ALCUNI ALGORITMI DI COMPRESSIONE

Tesi di Laurea in Fisica Matematica

Relatore:
Prof. Mirko Degli Esposti

Presentata da:
Stefania Basili

Terza Sessione
Anno Accademico 2009-2010

..un Grazie di cuore
alla mia famiglia!

Introduzione

Un paradosso della moderna tecnologia informatica è che nonostante un aumento quasi incomprensibile della memorizzazione dei dati e delle capacità di trasmissione, ci si sforza sempre più ad usare la compressione per aumentare la quantità di dati che possono essere manipolati.

Nel corso degli anni sono stati inventati e reinventati molti metodi di compressione. Si va da numerose tecniche ad hoc a metodi più di principio che possono dare compressioni molto buone. Uno dei primi e tra i migliori metodi conosciuti per la compressione del testo è la codifica di Huffman, che usa lo stesso principio del codice Morse (in cui il simbolo più frequente è rappresentato da un singolo punto): simboli comuni sono codificati in pochi bit, mentre quelli più rari hanno parole di codice più lunghe. Pubblicato al principio del 1950, la codifica di Huffman venne considerata come uno dei migliori metodi di compressione per diversi decenni, fin quando, nel tardo 1970, due scoperte -la compressione Ziv-Lempel e la codifica aritmetica- produssero i più elevati tassi di compressione.

Definiremo dapprima la nozione di un codice istantaneo e poi proveremo l'importanza della disuguaglianza di Kraft. Calcoli elementari mostreranno, in seguito, che la durata della rappresentazione prevista deve essere maggiore o uguale all'entropia, uno dei più importanti risultati. La semplice costruzione di Shannon mostra che la durata della rappresentazione prevista può realizzare questo limite asintoticamente per le descrizioni ripetute. Questo fa

riconoscere l'entropia come una misura naturale della durata della rappresentazione efficiente. Viene fornita la famosa procedura di codofica di Huffman per trovare minime attribuzioni alla durata della rappresentazione prevista.

Mostreremo, quindi, che i codici di Huffman sono competitivamente ottimali e richiedono circa H lanci di una moneta equilibrata per generare un modello di una variabile aleatoria avente entropia H. Quindi l'entropia è il limite della compressione dei dati nonchè il numero di bit necessario per la generazione di numeri casuali, ed i codici che conseguono H si rivelano ottimali da molti punti di vista.

Alcuni dei migliori metodi di compressione disponibili sono varianti di una tecnica chiamata *prediction by partial matching* (PPM), che fu sviluppata nel 1980. PPM si basa sulla codifica aritmetica per ottenere un buon rendimento di compressione.

Indice

Capitolo 1

Tecniche di Compressione

Esistono diversi algoritmi di compressione, alcuni sono generali e offrono prestazioni simili su molte varietà di dati, altri sono più mirati e in genere permettono compressioni superiori e presuppongono che l'utente conosca in anticipo il tipo di dati da comprimere. Le prestazioni della compressione dati dipendono da ciò che sappiamo delle caratteristiche del file sorgente, quindi dato un file, le sue caratteristiche possono essere usate per migliorare la compressione della sua stringa in output. Quando queste caratteristiche sono determinate prima della compressione, questa è detta conoscenza *a priori* delle caratteristiche dei dati da comprimere, che è utile per ottenere algoritmi di compressione più efficienti. Ad ogni modo, nella maggior parte dei casi non possiamo avere una conoscenza a priori delle caratteristiche del file, cosa che accade spesso nelle applicazioni reali.

I diversi algoritmi di compressione possono essere raggruppati in due grandi categorie:

- Compressione statistica
  (di cui fanno farte codifiche come quella di Huffman)
- Compressione mediante sostituzione di testo o con dizionario
  (un esempio sono gli algoritmi di compressione Ziv-Lempel)

1.1 Compressione dei dati

Immaginiamo di dover trasportare dei palloncini per addobbare una sala, se sono già gonfi è praticamente impossibile trasportarli su un'automobile, bisognerà necessariamente sgonfiarli prima di caricarli, e rigonfiarli nuovamente una volta giunti a destinazione. Quello appena illustrato altro non è che il concetto di *compressione dati*, basta sostituire i palloncini con programmi o file e la macchina con gli strumenti di internet o con i dispositivi di memorizzazione, veloci ma limitati per quanto riguarda lo 'spazio' (o la velocità di trasmissione) a disposizione.

Il termine compressione dati si riferisce al processo di trasformare dei dati eliminandone le informazioni ridondanti.
Da questa forma ridotta ottenuta, si riuscirà poi a ricostruire i dati originali o qualche buona approssimazione di essi.

La compressione dati ha molti usi. Nell'ambito delle reti il tempo di trasferimento dei file può essere significativamente ridotto comprimendo file di grandi dimensioni. Questo potrebbe essere utile per recuperare informazioni da database o per scaricare immagini. Inoltre, la compressione è utile anche per codificare i dati in modo che occupino meno spazio di quello che occupavano in origine o per effettuare il back up di file, riducendo così lo spazio richiesto per la memorizzazione delle informazioni.

Un ulteriore esempio di compressione è la trasmissione di dati tramite FAX, in cui lo scanner converte la pagina in sequenze di bit, che indicano dove si trovano le aree di scritto e dove quelle bianche. In genere si usa una risoluzione abbastanza elevata, perciò una semplice A diventa una lunga sequenza di simboli uguali. Poniamo inoltre il caso in cui lo scanner sia a colori, ogni punto sarebbe rappresentato da una tripla di numeri che indicano le intensità di colore RGB (Red, Green, Blue).
Ovviamente memorizzare in questo modo la prima riga occuperebbe alcuni

byte. Questa stessa riga potrebbe essere espressa in modo più compatto elencando semplicemente le variazioni di colore, per esempio con il formato: N(RGB), dove N rappresenta il numero di punti consecutivi con il colore (RGB).

Quindi la compressione dei dati è di notevole importanza per la trasmissione a distanza, in quanto alla riduzione dei dati si può accompagnare una utile riduzione della banda di trasmissione (avere un minor numero di dati in un certo intervallo di tempo comporta, infatti, una banda di trasmissione minore).

Con alcuni metodi si ottiene un'elevata compressione, formando buoni modelli di dati che devono essere codificati. La funzione di un modello è di *predire* simboli, che equivale a fornire una distribuzione di probabilità per il successivo simbolo che deve essere codificato. L'insieme di tutti i simboli possibili è chiamato *alfabeto* e, in esso, la distribuzione di probabilità fornisce una probabilità stimata per ogni simbolo.

Il modello fornisce questa distribuzione di probabilità al codificatore, il quale la usa per codificare il simbolo che effettivamente si verifica. Il decodificatore usa un modello identico insieme al rendimento del codificatore per scoprire qual è stato il simbolo codificato.

Il numero di bit in cui un simbolo, s, dovrebbe essere codificato è chiamato *indice di informazione*, $I(s)$ del simbolo, ed è direttamente collegato alla sua probabilità prevista, $Pr\,[s]$, dalla funzione $I(s) = -logPr\,[s]$ bit.
La quantità media di informazione per simbolo su tutto l'alfabeto è conosciuta come l'*entropia* della distribuzione di probabilità, denotata da H.
E' data da $H = \sum_s Pr[s] \cdot I(s) = \sum_s -Pr[s] \cdot logPr[s]$.
H è il limite inferiore nella compressione, misurato in bit per simbolo, che può essere raggiunto da ogni metodo di codifica.

1.2 Entropia ed ottimalità secondo Shannon

Uno dei primi ad occuparsi del problema della compressione fu Claude Elwood Shannon, che pose le basi della sua teoria della codifica, tuttora un caposaldo per chiunque si interessi di compressione. Nella sua teoria Shannon cerca una misura della quantità detta *entropia*. In generale, questo termine rappresenta il grado di disordine di un qualcosa; in termodinamica per esempio quantifica la dispersione di energia nell'universo. Nella teoria dell'informazione è misura inversa della comprimibilità di un messaggio, ed è in funzione delle probabilità di comparire dei simboli dell'alfabeto considerato. Chiamando la probabilità dell' i-esimo simbolo p_i e la cardinalità dell'insieme di simboli n, l'entropia deve quindi essere funzione degli argomenti $p_1...p_n$ compresi tra zero ed uno tali che la loro somma sia uno. Si presuppone inoltre che la sequenza di simboli che forma il messaggio sia ergodica; definiremo ergodica una sequenza dove ogni sottosequenza abbastanza lunga di caratteri goda delle stesse proprietà statistiche. Shannon impone che la funzione dell'entropia H debba avere le seguenti proprietà:

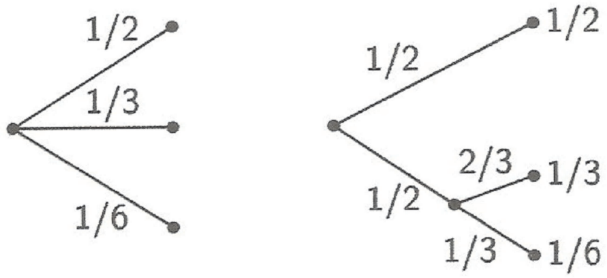
1. Continuità: H deve dipendere con continuità dai suoi argomenti p_i

2. Se gli argomenti valgono tutti $1/n$, l'entropia deve essere una funzione monotona crescente in n. Questo perchè se i simboli compaiono con uguale probabilità, un incremento del loro numero porta ad avere un messaggio maggiormente caotico.

3. Se dei simboli vengono raggruppati in un gruppo B, l'entropia è data da $H(p_1...p_n) = H(p_i...p_B) + p(B)H((p_j|B)...) \ \ \forall i \notin B \wedge \forall j \in B$

Dove nel primo addendo del membro destro dell'equazione l'entropia ha come argomenti la probabilità del gruppo p_B e tutti i p_i tali che $i \notin B$ e nel secondo addendo la funzione H è applicata a tutti i $p_j|B$ tali che $j \in B$. Quest'ultima regola definisce come si deve comportare la funzione se la probabilità di un simbolo è vista come probabilità di scegliere un gruppo che lo contenga moltiplicata la probabilità che, una volta che il suo gruppo è stato

selezionato, venga scelto il simbolo stesso. Detta anche proprietà di raggruppamento, questa caratteristica assicura coerenza alla funzione, garantendo che il valore di H non cambi applicando viste diverse su dati uguali. Shannon dà come esempio della proprietà di raggruppamento la seguente figura



$$H(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) = H(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{2}{3},\tfrac{1}{3})$$

Figura 1.1: Esempio di raggruppamento dell'entropia

.

Presupponendo che i simboli siano indipendenti l'uno dall'altro, l'unica famiglia di funzioni che soddisfa le precedenti è rappresentata dalla formula

$$-k\sum_{i=1}^{n} log_2 p_i, \quad k > 0.$$

Dove k è un fattore di proporzionalità che determina l'unità di misura, e viene per convenzione imposto uguale ad uno. Shannon ha dimostrato che tale quantità limita in modo rigido la compressione che è possibile ottenere. Infatti, un messaggio non è mai comprimibile sotto il limite di H bit per carattere; per questo l'entropia è talvolta misurata in bit per simbolo.
Con l'ipotesi dell'indipendenza statistica fra i simboli, abbiamo realizzato quello che si dice modello di ordine uno. Se consideriamo inoltre che i caratteri siano equiprobabili otteniamo il modello di ordine zero; tale modello rispecchia l'assoluta casualità ed è regolato dalla formula $H = log_2 n$, il valore massimo dell'entropia per ogni n fissato.

In generale, ogni modello di un determinato ordine comprende tutti quelli di ordine minore.
Nel modello di ordine due si tiene conto del legame che ci può essere tra caratteri adiacenti, sostituendo a p_i una probabilità in dipendenza dal suo conesto; la formula dell'entropia sarà la seguente:

$$\sum_{j=1}^{n} p_j \ \sum_{i=1}^{n} p_{i|j} \ log_2 \ p_{i|j}.$$

In generale, possiamo ottenere un generico modello di ordine n considerando la probabilità di p_i in dipendenza da altri $n-1$ caratteri che lo precedono. Al crescere dell'ordine del modello, teniamo conto di maggiori informazioni, e troveremo quindi valori dell'entropia decrescenti. Il vero valore dell'entropia di un messaggio è dato dal modello generale ottenuto considerando il limite di un modello di ordine n. Tale quantità è chiaramente incalcolabile; quello che si può fare è calcolare l'entropia per un modello che la approssimi sufficientemente bene. In messaggi reali è in effetti difficile limitare la quantità di caratteri di un contesto che ci danno informazioni sul simbolo successivo, ma è altrettanto vero che il contributo dato dai simboli più distanti solitamente è quasi nullo e che quindi l'ignorarli non provoca gravi errori.

Una grande utilità della teoria di Shannon è che questa ci suggerisce il numero di bit da utilizzare per codificare un determinato simbolo. Un messaggio viene infatti compresso alla lunghezza minima di H bit per carattere utilizzando esattamente $log_2 \ p_i$ bit per la codifica di ogni simbolo. I principali problemi da affrontare per arrivare all'ottimalità sono quindi due:

- nella maggior parte dei casi, la quantità $log_2 \ p_i$ non è un numero intero. Ci troviamo quindi di fronte alla difficoltà di dover allocare un numero frazionario di bit per simbolo. Una prima soluzione è quella di usare un metodo che costruisca in qualche modo un insieme di caratteri tali che ogni valore $log_2 \ p_i$ sia un numero elevato minimizzando così l'errore relativo fatto arrotondandolo all'intero più vicino.

Shannon per esempio suggerisce di minimizzare l'errore raggruppando i simboli dell'alfabeto a gruppi di n ottenendo una distribuzione di probabilità più granulare con possibilità di apparire minore per ogni ennupla di caratteri; tale algoritmo non può essere applicato tuttavia a grandi gruppi in quanto la sua complessità è esponenziale. Un altro metodo, rappresentato dalla codifica aritmetica, riesce effettivamente ad utilizzare un numero non intero di bit per simbolo, anche se al costo di una complessità computazionale piuttosto alta in fase di codifica e decodifica.

- la quantità p_i è difficile da calcolare in quanto al crescere dell'ordine del modello la complessità del calcolo delle probabilità dei simboli sale in modo esponenziale. Solitamente questo problema è di più difficile risoluzione del precedente; un algoritmo capace di sfruttare anche modelli di ordine elevato per calcolare queste quantità è il PPM, che verrà descritto in seguito.

1.3 Definizioni e prime proprietà

Definizione 1.1. Un *codice* C per una variabile aleatoria X è una mappa $\chi \longrightarrow D^*$ dove D è un insieme di cardinalità D (insieme dei simboli) e $D^* = \bigcup_n D^n$, insieme di tutte le parole di tutte le lunghezze che è possibile formare con i simboli di D. Si denota con $C(x)$ la *parola* (di codice) o *traduzione* di x e con $l(x)$ la *lunghezza* di $C(x)$.

Per esempio, $C(red) = 00$, $C(blue) = 11$ è un codice per $\chi = \{red, blue\}$ con l'alfabeto $D = \{0, 1\}$.

Definizione 1.2. La *lunghezza media* $L(C)$ del codice $C(x)$ per una variabile aleatoria X con la funzione di probabilità $p(x)$ è data da $L(C) = \sum_{x \in \chi} p(x)\, l(x)$, dove $l(x)$ è la lunghezza di $C(x)$.
Senza perdita di generalità, possiamo assumere che l'alfabeto D-ario è $D = \{0, 1, ..., D-1\}$.

Esempio 1.1. Sia X una variabile aleatoria con la seguente distribuzione e il relativo codice:
$Pr(X = 1) = 1/2$, $C(1) = 0$
$Pr(X = 2) = 1/4$, $C(2) = 10$
$Pr(X = 3) = 1/8$, $C(3) = 110$
$Pr(X = 4) = 1/8$, $C(4) = 111$
L'entropia $H(X)$ di X è $7/4 = 1.75$ bit, e la lunghezza media $L(C)$ di questo codice è anche $7/4 = 1.75$ bit. Qui si ha un codice che ha la stessa lunghezza media dell' entropia. Si nota che ogni sequenza di bit può essere univocamente decodificata in una sequenza di simboli di X.
Per esempio, la stringa di bit 0110111100110 è decodificata come 134213.

Esempio 1.2. (*Codice Morse*) Il Codice Morse è un codice ragionevolmente efficiente per l'alfabeto Inglese poichè usa un alfabeto di quattro simboli: un punto, un trattino, uno spazio di lettera e uno spazio di parola. Le sequenze corte rappresentano lettere frequenti (un singolo punto rappresenta E),
mentre le sequenze lunghe rappresentano lettere rare (Q è rappresentata da 'trattino,trattino,punto,trattino'). Questa non è la rappresentazione ottimale per l'alfabeto in quattro simboli: molti codici possibili, infatti, non vengono utilizzati perchè i codici per le lettere non contengono spazi eccetto che per uno spazio di lettera alla fine di ogni codice, e nessuno spazio può seguire un altro spazio. Questo è un problema interessante per calcolare il numero delle sequenze che possono essere costruite sotto questi vincoli. Il problema fu risolto da Shannon nel 1948 nel suo originale saggio.

Definizione 1.3. C è detto *non singolare* se è iniettivo, cioè se $x \neq x^{'}$
$\Rightarrow C(x) \neq C(x^{'})$.

Definizione 1.4. L'*estensione* C^* di un codice C è la mappa da stringhe di lunghezza finita di χ a stringhe sempre di lunghezza finita di D,
$C^* : \chi^* \rightarrow D^*$, definita da $C(x_1x_2 \cdots x_n) = C(x_1)C(x_2) \cdots C(x_n)$, dove $C(x_1)C(x_2) \cdots C(x_n)$ indica la concatenazione dei codici corrispondenti.

Per esempio, se $C(x_1) = 00$ e $C(x_2) = 11$, allora $C(x_1x_2) = 0011$.

Definizione 1.5. C è *univocamente decifrabile* se la sua estensione C^* è non singolare. In altre parole, qualsiasi stringa codificata in un codice univocamente decifrabile ha solo una possibile stringa che la produce. Comunque, si può dare un'occhiata all'intera stringa per determinare anche il primo simbolo in quella corrispondente.

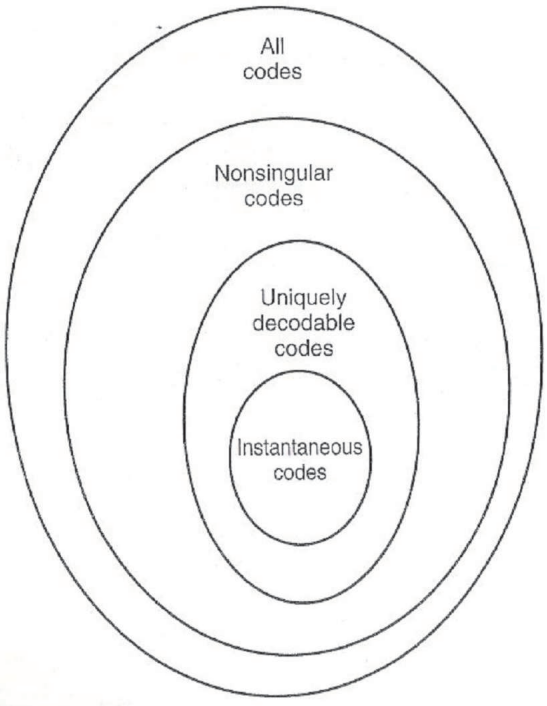
Definizione 1.6. Un codice è detto *istantaneo* se nessuna parola di codice è prefisso di un'altra parola.
Un tale codice può essere decodificato senza fare riferimento alle parole di codice future poichè la fine di una parola è immediatamente riconoscibile, quindi, il simbolo x_i può essere decodificato non appena arriviamo alla fine della parola di codice ad esso corrispondente, senza aspettare di vedere le altre che verranno dopo. Un codice istantaneo è un *codice di auto-punteggiatura*; siamo in grado di leggere da sinistra a destra la sequenza dei simboli del codice e aggiungere le virgole per separare le parole senza guardare i simboli successivi.
Per esempio, la stringa binaria 01011111010 prodotta dal codice dell' Esempio 1.1 viene analizzata come $0, 10, 111, 110, 10$.

Per illustrare le differenze tra i vari tipi di codici, consideriamo la seguente tabella:

X	singolare	non singolare,ma non univocamente decifrabile	univocamente decifrabile, ma non istantaneo	istantaneo
1	0	0	10	0
2	0	010	00	10
3	0	01	11	110
4	0	10	110	111

Per il codice non singolare, la stringa 010 ha tre possibili sequenze: 2 o 14 o 31, quindi non è univocamente decifrabile. Il codice univocamente decifrabile non è libero da prefissi e quindi non è istantaneo.
Per vedere che è univocamente decifrabile, si prende ogni stringa e si comincia dall'inizio. Se i primi due bit sono 00 o 10, essi possono essere codificati immediatamente. Se invece sono 11, dobbiamo guardare il bit successivo. Se è 1, il primo simbolo è 3. Se la lunghezza della stringa di 0 subito dopo 11 è dispari, il primo codice deve essere 110 ed il primo simbolo 4; se invece è pari, il primo simbolo è 3. Ripetendo questo ragionamento, possiamo vedere che il codice è univocamente decifrabile. Il fatto che l'ultimo codice nella tabella è istantaneo è ovvio poichè nessuno è prefisso di un altro.

L'andamento di queste definizioni è mostrato nella seguente figura



1.4 Disuguaglianza di Kraft

Vogliamo costruire codici istantanei di lunghezza media minima per descrivere una determinata fonte. E' chiaro che non possiamo assegnare brevi parole di codice a tutti i simboli e che risultino ancora istantanei. L'insieme delle lunghezze delle parole di codice possibili per i codici istantanei è limitato dalla seguente disuguaglianza [1].

Teorema 1.4.1. (Disuguaglianza di Kraft)
Dato un codice istantaneo su un alfabeto di cardinalità D e con lunghezze $l_1, l_2, ..., l_m$, *si ha* $\sum_i D^{-l_i} \leq 1$.
Viceversa, se abbiamo una famiglia di lunghezze del codice $\{l_1, ..., l_m\} \subset Z^+$ *che soddisfa tale disuguaglianza, allora esiste un codice istantaneo D-ario che ha le* l_i *come lunghezze (della parola di codice).*

Dimostrazione. Consideriamo un albero D-ario in cui ogni nodo ha D discendenti. Siano i rami dell'albero i simboli della parola di codice; per esempio, i D rami derivanti dal nodo della radice rappresentano i D valori possibili del primo simbolo della parola di codice. In seguito ogni codice è rappresentato da una foglia nell'albero. Il percorso dalla radice traccia i simboli del codice. Consideriamo ora un alfabeto di cardinalità 2; la condizione di codice istantaneo implica che nessun codice è un ascendente di un altro nell'albero. Quindi, ogni codice elimina il suo discendente come codice possibile.
Sia l_{max} la lunghezza del codice più lungo. Consideriamo tutti i nodi dell'albero al livello l_{max}. Alcuni di essi sono codici, alcuni sono discendenti di codici , e altri nessuno dei due. Un codice al livello l_i ha $D^{l_{max}-l_i}$ discendenti al livello l_{max}. Ciascuno di questi gruppi discendente deve essere disgiunto. Inoltre, il numero totale di nodi in questi gruppi deve essere minore o uguale a $D^{l_{max}}$, si ha quindi: $\sum D^{l_{max}-l_i} \leq D^{l_{max}}$, cioè $\sum D^{-l_i} \leq 1$, nonchè la Disuguaglianza di Kraft.
Viceversa, date le lunghezze l_i che soddisfano la disuguaglianza, è sempre possibile costruire il codice (l'albero).

Classifichiamo il primo nodo di profondità l_1 come codice 1 ed eliminiamo dall'albero i suoi discendenti.
Poi classifichiamo il primo nodo rimanente di profondità l_2 come codice 2, e così via. Procedendo in questo modo, costruiamo un codice istantaneo con le lunghezze l_i specificate. □

Quindi, ogni codice che soddisfa la condizione di istantaneo deve soddisfare anche la disuguaglianza di Kraft, che è condizione sufficiente per l'esistenza di un codice con le lunghezze specificate.
Consideriamo ora il problema di trovare il codice istantaneo con la minima lunghezza media. Ciò equivale a trovare l'insieme di lunghezze $l_1, l_2, ..., l_m$ che soddisfano la disuguaglianza di Kraft e la cui lunghezza media $L = \sum p_i\ l_i$ è minore della lunghezza media di ogni altro codice istantaneo.
Questo è un problema standard di ottimizzazione:
lo scopo del gioco è minimizzare L con il vincolo $\sum D^{-l_i} \leq 1$.
Siano $l_1^*, ..., l_m^*$ i punti di minimo. Trascuriamo il vincolo di intero su l_i e assumiamo l'uguaglianza nel vincolo. Possiamo quindi scrivere la minimizzazione vincolata usando i moltiplicatori di Lagrange come la minimizzazione di $J = \sum p_i\ l_i + \lambda\ (\sum D^{-l_i})$.
Differenziando rispetto a l_i, si ottiene $\dfrac{\partial J}{\partial l_i} = p_i - \lambda\ D^{-l_i}\ log_e\ D.$

Facendo tendere la derivata a 0, si ha $D^{-l_i} = \dfrac{p_i}{\lambda\ log_e\ D}.$

Sostituendo poi nel vincolo per trovare λ, si ha $\lambda = \dfrac{1}{log_e\ D}$ e quindi

$p_i = D^{-l_i}$, fornendo le lunghezze del codice ottimale, $l_i^* = -log_D\ p_i$ (caso superottimale). Questa scelta delle lunghezze non intere della parola del codice fornisce la lunghezza media del codice
$L^* = L(C) = \sum p_i\ l_i^* = -\sum p_i\ log_D\ p_i = H_D(X)$
(Nel caso superottimale, $L^* = H_D(X)$ è l'entropia D-aria della variabile).
Ma, poichè le l_i devono essere intere, dovremmo scegliere un insieme di lunghezze del codice vicino a quello ottimale.

Verifichiamo l'ottimalità direttamente nella dimostrazione del seguente teorema.

Teorema 1.4.2. *La lunghezza media L di ogni codice istantaneo D-ario per una variabile aleatoria X è maggiore o uguale all'entropia $H_D(X)$, cioè $L \geq H_D(X)$. L'uguaglianza vale se e solo se $p_i = D^{-l_i}$, dove l_i è la lunghezza della parola di codice associata ad $i \in X$.*

Dimostrazione.
Possiamo scrivere la differenza tra la lunghezza media e l'entropia come

$$L - H_D(X) = \sum p_i \, l_i - \sum p_i \, log_D \, \frac{1}{p_i} = - \sum p_i \, log_D \, D^{-l_i} + \sum p_i \, log_D \, p_i.$$

Siano $r_i = \dfrac{D^{-l_i}}{\sum_j D^{-l_j}}$ e $c = \sum D^{-l_i}$, otteniamo quindi

$$L - H = \sum p_i \, log_D \, \frac{p_i}{r_i} - log_D \, c = D(\mathbf{p} \| \mathbf{r}) + log_D \, \frac{1}{c}$$

dalla non-negatività dell'entropia relativa e dal fatto che $c \leq 1$ (disuguaglianza di Kraft), quindi $L \geq H$ (vale l'uguaglianza se e solo se $p_i = D^{-l_i}$, cioè se e solo se $-log_D \, p_i$ è un intero per ogni i). □

Definizione 1.7. Una distribuzione di probabilità è detta *D-adica* se ognuna delle probabilità è pari a D^{-n} per qualche n. Perciò, nel teorema si ha l'uguaglianza se e solo se la distribuzione di X è D-adica.

Osservazione: La superottimalità si ha se e solo se le probabilità p_i di X sono D-adiche.

La dimostrazione precedente mostra anche una procedura per trovare un codice ottimale: trovare la distribuzione D-adica che è più vicina (nel senso di entropia relativa) alla distribuzione di X. Questa distribuzione fornisce l'insieme delle lunghezze del codice. Costruire il codice scegliendo il primo nodo disponibile come nella dimostrazione della disuguaglianza di Kraft.
Abbiamo quindi un codice ottimale per X. Comunque, questa procedura

non è facile, poichè la ricerca per la distribuzione D-adica più vicina non è evidente.

Diamo ora una buona procedura sub-ottimale, la codifica di Shannon-Fano. Mostriamo un codice che realizza una lunghezza media L della descrizione entro 1 bit del limite inferiore; cioè, $H(X) \leq L < H(X) + 1$.
Vogliamo minimizzare $L = \sum p_i \, l_i$ soggetto al fatto che $l_1, l_2, ..., l_m$ sono interi e vincolati da $\sum D^{-l_i} \leq 1$.
Proviamo che le lunghezze del codice ottimale possono essere trovate cercando la distribuzione di probabilità D-adica più vicina alla distribuzione di X nell'entropia relativa, ossia, cercando $\mathbf{r}$ $\left(r_i = \dfrac{D^{-l_i}}{\sum\limits_j D^{-l_j}}\right)$ e minimizzando

$L - H_D = D(\mathbf{p}\|\mathbf{r}) - log(\sum D^{-l_i}) \geq 0.$

La scelta delle lunghezze $l_i = log_D \frac{1}{p_i}$ fornisce $L = H$.
Poichè $log_D \frac{1}{p_i}$ potrebbe non essere un intero, arrotondiamo per eccesso fino ad ottenere assegnazioni di lunghezze intere, $l_i = \lceil log_D \dfrac{1}{p_i} \rceil$, dove $\lceil x \rceil$ è il più piccolo intero $\geq x$.
Queste lunghezze soddisfano la disuguaglianza di Kraft, infatti

$$\sum D^{-\lceil log \frac{1}{p_i} \rceil} \leq \sum D^{-log \frac{1}{p_i}} = \sum p_i = 1.$$

Questa scelta di lunghezze soddisfa $log_D \dfrac{1}{p_i} \leq l_i < log_D \dfrac{1}{p_i} + 1.$

Moltiplicando per p_i e sommando su i, si ottiene $H_D(X) \leq L < H_D(X) + 1$.
Dal momento che un codice ottimale può solo essere migliore di questo, si ha il seguente teorema.

Teorema 1.4.3. *Siano $l_1^*, l_2^*, ..., l_m^*$ le lunghezze del codice ottimale per una distribuzione $\boldsymbol{p}$ ed un alfabeto D-ario, e sia L^* la sua lunghezza media associata ($L^* = \sum p_i l_i^*$). Allora esiste un codice istantaneo C tale che*
$H_D(X) \leq L^* < H_D(X) + 1$.

Dimostrazione. Sia $l_i = \lceil log_D \frac{1}{p_i} \rceil$ il *Codice di Shannon*. Allora le l_i soddisfano la disuguaglianza di Kraft e si ha $H_D(X) \leq L = \sum p_i \, l_i < H_D(X) + 1$. Ma poichè L^*, la lunghezza media del codice ottimale, è minore di $L = \sum p_i \, l_i$, e maggiore o uguale a H_D per il Teorema 1.2.2, si ha la tesi.

□

Consideriamo un sistema nel quale inviamo da X una sequenza di n simboli, che assumiamo i.i.d. conformemente a $p(x)$. Possiamo considerare questi n simboli come supersimboli dall'alfabeto X^n.
Sia L_n la lunghezza media del codice per il simbolo di input, cioè, se $l(x_1, x_2, ..., x_n)$ è la lunghezza del codice binario associato con $x_1, x_2, ..., x_n$ (d'ora in poi assumiamo che $D = 2$ per semplicità), allora

$$L_n = \frac{1}{n} \sum p(x_1, x_2, ..., x_n) l(x_1, x_2, ..., x_n) = \frac{1}{n} El(X_1, X_2, ..., X_n).$$

Possiamo ora applicare al codice le derivate dei limiti:

$$H(X_1, X_2, ..., X_n) \leq El(X_1, X_2, ..., X_n) < H(X_1, X_2, ..., X_n) + 1.$$

Poichè $X_1, X_2, ..., X_n$ sono i.i.d., $H(X_1, X_2, ..., X_n) = \sum H(X_i) = nH(X)$.

Dividendo per n, si ottiene $H(X) \leq L_n < H(X) + \frac{1}{n}$.

Quindi, usando grandi blocchi di lunghezze, possiamo ottenere una lunghezza media del codice per simbolo arbitrariamente vicina all' entropia.

Possiamo fare lo stesso ragionamento per una sequenza di simboli da un processo stocastico che non sia necessariamente i.i.d..

In questo caso, abbiamo ancora il limite
$H(X_1, X_2, ..., X_n) \leq El(X_1, X_2, ..., X_n) < H(X_1, X_2, ..., X_n) + 1.$

Dividendo ancora per n ed essendo L_n la lunghezza media per simbolo,

otteniamo $\frac{H(X_1, X_2, ..., X_n)}{n} \leq L_n < \frac{H(X_1, X_2, ..., X_n)}{n} + \frac{1}{n}.$

Se il processo stocastico è stazionario, allora $\frac{H(X_1, X_2, ..., X_n)}{n} \to H(\chi),$

e la lunghezza media tende al tasso di entropia come $n \to \infty$.
Perciò si ha il seguente teorema:

Teorema 1.4.4. *La minima lunghezza media del codice per simbolo soddisfa*

$$\frac{H(X_1, X_2, ..., X_n)}{n} \leq L_n^* < \frac{H(X_1, X_2, ..., X_n)}{n} + \frac{1}{n}.$$

Comunque, se $X_1, X_2, ..., X_n$ è un processo stocastico stazionario,
$L_n^* \to H(\chi)$, dove $H(\chi)$ è il tasso di entropia del processo.

Questo teorema fornisce un'altra ragione per la definizione del tasso di entropia; esso è il numero medio di bit per simbolo necessario per descrivere il processo.

Finalmente, ci chiediamo cosa succede alla lunghezza media dell'informazione se il codice è stato progettato per la distribuzione sbagliata. Per esempio, la distribuzione sbagliata può essere la miglior stima che possiamo fare di quella reale sconosciuta. Consideriamo l'assegnazione del codice di Shannon $l(x) = \lceil log\frac{1}{q(x)} \rceil$ costruito per la funzione di probabilità $q(x)$.
Supponiamo che la vera funzione di probabilità sia $p(x)$.
Quindi, non realizzeremo la lunghezza media $L \approx H(p) = -\sum p(x) logp(x)$.

Mostriamo ora che l'incremento nella lunghezza media, dovuto alla distribuzione inesatta, è l'entropia relativa $D(p\|q)$. Quindi, $D(p\|q)$ è interpretato come l'incremento nella complessità descrittiva dovuto all'informazione inesatta.

Teorema 1.4.5. (Codice errato)
Se X è distribuito secondo $p(x)$ e il codice è quello di Shannon ottimizzato per una presunta distribuzione $q(x)$ ($l(x) = \lceil log\frac{1}{q(x)} \rceil$), allora
$H(p) + D(p\|q) \leq L(C) < H(p) + D(p\|q) + 1$.

Dimostrazione. La lunghezza media del codice è

$$L(C) = \sum_x p(x)\ l(x) = \sum_x p(x)\ \lceil log\ \frac{1}{q(x)} \rceil$$

$$< \sum_x p(x)\ (log\ \frac{1}{q(x)} + 1)$$

$$= \sum_x p(x)\ log\ \frac{p(x)}{q(x)}\ \frac{1}{p(x)} + 1$$

$$= \sum_x p(x)\ log\ \frac{p(x)}{q(x)} + \sum_x p(x)\ log\ \frac{1}{p(x)} + 1$$

$$= D(p\|q) + H(p) + 1.$$

Il limite inferiore si ottiene allo stesso modo.

□

Quindi, supponendo che la distribuzione sia $q(x)$ quando quella vera è $p(x)$ si incorre ad una penalità di $D(p\|q)$ nella lunghezza media.

1.5 Disuguaglianza di Kraft per codici univocamente decifrabili

Abbiamo provato che ogni codice istantaneo deve soddisfare la disuguaglianza di Kraft. L'insieme dei codici univocamente decifrabili è più grande dell'insieme di quelli istantanei, così ci si aspetta di conseguire una lunghezza media più bassa se L è minimizzata su tutti i codici univocamente decifrabili. In questa sezione proveremo che l'insieme di tali codici non offre alcuna possibilità ulteriore per l'insieme delle lunghezze più di quanto non facciano i codici istantanei. Diamo ora l'elegante dimostrazione di Karush del seguente teorema.

Teorema 1.5.1. (McMillan)
Le lunghezze $l_1, ..., l_m$ di un codice univocamente decifrabile soddisfano la disuguaglianza di Kraft $\sum D^{-l_i} \leq 1$.
Viceversa, dato un insieme di lunghezze di codice che soddisfano tale disuguaglianza, è possibile costruire un codice univocamente decifrabile con queste lunghezze.

Dimostrazione. Consideriamo C^k, l' estensione k-esima del codice C. Dalla definizione di decifrabilità unica, la k-esima estensione del codice è non singolare. Poichè ci sono solo D^n stringhe diverse di lunghezza n, la decifrabilità unica implica che il numero delle sequenze di codice di lunghezza n nell' estensione k-esima deve essere più grande di D^n. Usiamo quindi questa affermazione per provare la disuguaglianza di Kraft.
Siano le lunghezze del codice dei simboli $x \in \chi$ denotate da $l(x)$. Per l' estensione, la lunghezza della sequenza del codice è $l(x_1, x_2, ..., x_k) = \sum_{i=1}^{k} l(x_i)$.
La disuguaglianza che vogliamo provare è $\sum_{x \in \chi} D^{-l(x)} \leq 1$.

Il trucco è quello di considerare la potenza k-esima di questa quantità.

Quindi, $\big(\sum_{x\in\chi} D^{-l(x)}\big)^k = \sum_{x_1\in\chi}\sum_{x_2\in\chi}\cdots\sum_{x_k\in\chi} D^{-l(x_1)}\ D^{-l(x_2)}\ \cdots\ D^{-l(x_k)}$

$= \sum_{x_1,x_2,\ldots,x_k\in\chi^k} D^{-l(x_1)}\ D^{-l(x_2)}\ \cdots\ D^{-l(x_k)} = \sum_{x^k\in\chi^k} D^{-l(x^k)}.$

Raccogliamo ora i termini delle lunghezze della parola per ottenere

$$\sum_{x^k\in\chi^k} D^{-l(x^k)} = \sum_{m=1}^{kl_{max}} a(m)\ D^{-m},$$

dove l_{max} è la lunghezza massima del codice e $a(m)$ è il numero delle sequenze x^k in codici di lunghezza m.
Il codice è univocamente decifrabile, così c'è almeno una sequenza che va in ogni m-sequenza del codice e almeno D^m m-sequenze.
Quindi, $a(m) \leq D^m$, e si ha

$$\big(\sum_{x\in\chi} D^{-l(x)}\big)^k \ = \ \sum_{m=1}^{kl_{max}} a(m)\ D^{-m} \ \leq \ \sum_{m=1}^{kl_{max}} D^m\ D^{-m} \ = \ kl_{max}$$

e quindi $\sum_j D^{-l_j} \leq (kl_{max})^{1/k}$.

Poichè questa disuguaglianza è verificata per ogni k, vale anche per $k \to \infty$. Siccome $(kl_{max})^{1/k} \to 1$ si ottiene $\sum_j D^{-l_j} \leq 1$, che è la disuguaglianza di Kraft.
Viceversa, dato un qualsiasi insieme di lunghezze $l_1, l_2, ..., l_m$ soddisfacente la disuguaglianza, possiamo costruire un codice istantaneo. Poichè ogni codice istantaneo è univocamente decifrabile, si ha la tesi.

□

Capitolo 2

Metodi di codifica

La codifica ha il compito di determinare la rappresentazione di output di un simbolo, basata sulla distribuzione di probabilità fornita da un modello. L'idea generale è che un codificatore dovrebbe produrre codici corti per i simboli probabili e codici lunghi per quelli più rari. Ci sono limiti teorici su quanto può essere corta la lunghezza media di una parola di codice per una data distribuzione di probabilità, ed è stato fatto un grande sforzo per trovare programmatori che consentano di raggiungere tale limite.
Un'altra importante considerazione è la velocità del codificatore, è necessaria una ragionevole quantità di calcoli per generare codici quasi ottimali.
Se la velocità è importante, potremmo usare un codificatore che sacrifica le prestazioni di compressione per ridurre la quantità di sforzo richiesta.
Per esempio, se ci sono 256 possibili simboli che possono essere codificati, potremmo usare un codificatore che rappresenti i 15 simboli più probabili in 4 bit ed i restanti in 12 bit. Il colmo per questo tipo di approssimazione è proprio quello di codificare tutti i simboli in 8 bit. Esso non fornisce alcuna compressione ma è molto veloce.
Descriveremo i due seguenti metodi di codifica: *Codifica di Huffman* e *Codifica Aritmetica* (tesi a raggiungere l'ottimalità definita dall'entropia di Shannon).

La codifica di Huffman tende ad essere più veloce di quella aritmetica, ma la codifica aritmetica è in grado di produrre una compressione che è vicina a quella ottimale data la distribuzione di probabilità fornita dal modello. Per ognuno di questi due tipi di codificatori, guarderemo dapprima al criterio secondo cui si raggiunge la compressione e poi daremo i dettagli di come essi vengono messi in pratica [2].

2.1 La codifica di Huffman

Grazie alla teoria di Shannon, dati una serie di simboli con determinate frequenze, siamo in grado di determinare il numero di bit da utilizzare per ogni carattere. Con un semplice algoritmo scoperto da Huffman, possiamo costruire un codice istantaneo ottimale (durata media più breve) per una data distribuzione. Gli alberi di Huffman sono, infatti, una struttura ideata per elaborare una codifica univocamente decifrabile dove ad ogni carattere viene associato un codice di tale lunghezza ottimale.
Proveremo che ogni altro codice per lo stesso alfabeto non può avere lunghezza media inferiore a quella del codice costruito dall'algoritmo.

Una proprietà che ci garantisce una corretta decodificabilità di un messaggio è la seguente:
Proprietà del prefisso: Una codifica gode di questa proprietà se nessun codice assegnato è prefisso di un altro codice.

E' chiaro che una codifica così composta permette in fase di decompressione di determinare un simbolo appena il suo codice è stato letto. Il modo più semplice per costruirne una con questa proprietà è quella di costruire un *trie* binario, un particolare tipo di albero binario dove soltanto le foglie contengono i valori delle chiavi.

Associando a questo punto i valori da codificare alle foglie ed i bit zero ed uno ai due archi uscenti da ogni nodo, si definisce come codifica di un determinato simbolo la sequenza di bit da percorrere per arrivare dalla radice al carattere interessato. Chiaramente una codifica così definita soddisfa la proprietà del prefisso, e si può considerare il trie come un albero decisionale da usare in fase di decodifica.

Esempio 2.1. Consideriamo una variabile aleatoria X nell'insieme $\chi = \{1, 2, 3, 4, 5\}$ con le rispettive probabilità $0.25, 0.25, 0.2, 0.15, 0.15$.
Ci aspettiamo che il codice binario ottimale per X abbia le parole di codice più lunghe assegnate ai simboli 4 e 5. Queste due lunghezze devono essere uguali, altrimenti possiamo eliminare un bit dalla parola di codice più lunga ed avere ancora un codice istantaneo, ma con una lunghezza media più corta. In generale, possiamo costruire un codice nel quale le due parole più lunghe differiscono solo nell'ultimo bit. Per questo codice, siamo in grado di combinare i simboli 4 e 5 in uno solo, con un'attribuzione di probabilità di 0.30. Procedendo in questo modo, combinando i due meno probabili in un unico simbolo, fin quando rimaniamo con uno solo, ed assegnando poi parole di codice, otteniamo la seguente tabella:

Lunghezza del codice	Codice	X	Probabilità				
2	01	1	0.25	0.3	0.45	0.55	1
2	10	2	0.25	0.25	0.3	0.45	
2	11	3	0.2	0.25	0.25		
3	000	4	0.15	0.2			
3	001	5	0.15				

Questo codice ha lunghezza media di 2.3 bit.

Esempio 2.2. Consideriamo un codice ternario per la stessa variabile aleatoria. Ora combiniamo i tre simboli meno probabili in un supersimbolo ed otteniamo la seguente tabella:

Codice	X	Probabilità		
1	1	0.25	0.5	1
2	2	0.25	0.25	
00	3	0.2	0.25	
01	4	0.15		
02	5	0.15		

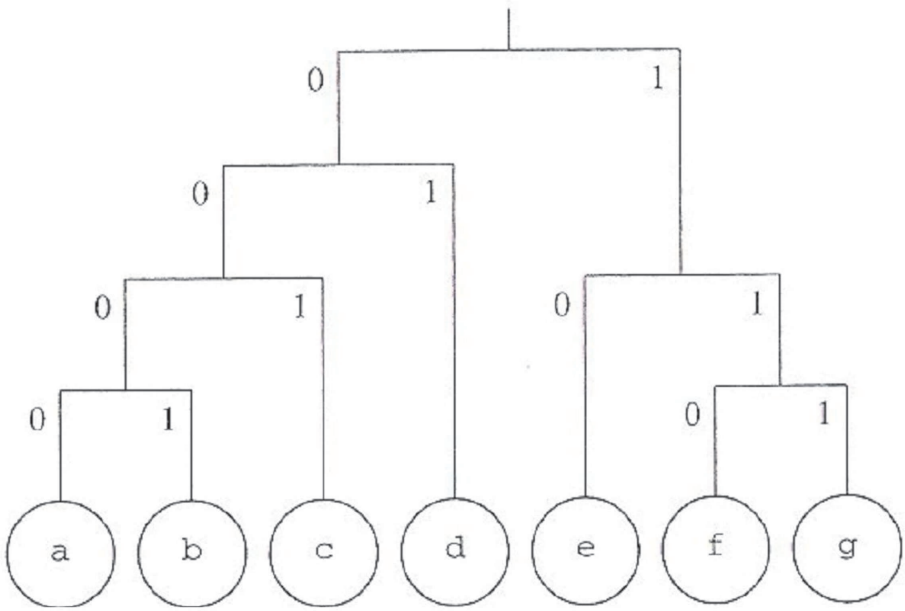
Questo codice ha una lunghezza media di 1.5 cifre ternarie.

La codifica di Huffman genera, quindi, parole di codice per un insieme di simboli, date alcune distribuzioni di probabilità. Le parole di codice generate forniscono la miglior compressione possibile per un codice istantaneo per la distribuzione di probabilità data.
La seguente tabella mostra i codici per i sette simboli dell'alfabeto a, b, c, d, e, f, g.

Simbolo	Codice	Probabilità
a	0000	0.05
b	0001	0.05
c	001	0.1
d	01	0.2
e	10	0.3
f	110	0.2
g	111	0.1

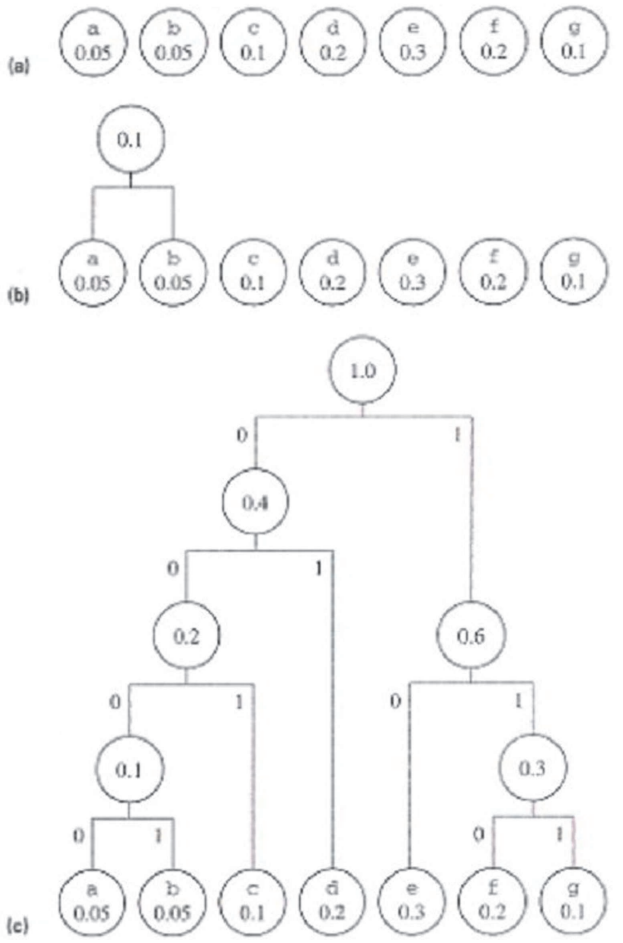
Una frase è codificata sostituendo ognuno dei suoi simboli con il codice dato dalla tabella. Per esempio, la frase $eefggfed$ è codificata come 10101101111111101001.

La decodifica viene eseguita da sinistra a destra. L'input per il decodificatore comincia con 10..., ed il solo codice che inizia in questo modo è quello per e, che viene quindi preso come primo simbolo.
La decodifica procede poi con la parte restante della stringa, 1011011....
La seguente figura mostra un albero che può essere usato per la decodifica



L'algoritmo di Huffman lavora costruendo l'albero di decodifica dal basso verso l'alto. Consideriamo, per esempio, l'insieme dei simboli con le rispettive probabilità mostrate nella tabella sopra; esso comincia creando per ogni simbolo un nodo di foglia contenente il simbolo e la sua probabilità (a). Quindi i due nodi con le probabilità minori diventano fratelli e sorelle nell'ambito di un nodo padre, il quale è dato da una probabilità pari alla somma di quelle dei suoi figli (b). Questa operazione di combinazione viene ripetuta, scegliendo i due nodi con le probabilità più piccole e ignorando quelli che sono già figli. Per esempio, al passo successivo il nuovo nodo formato combinando a e b si unisce con c per formarne uno nuovo con probabilità $p = 0.2$. Il processo continua fino a quando c'è solo un nodo senza parenti, che diventa quindi la

radice dell'albero di decodifica (c). I due rami vengono classificati con 0 e 1 (l'ordine non è importante) per formare l'albero.



Generalmente, la codifica di Huffman è veloce sia per codificare che per decodificare, purchè la distribuzione di probabilità sia statica. Comunque, i migliori modelli adattivi di solito usano molte diverse distribuzioni di probabilità allo stesso tempo, essendo la distribuzione appropriata stata scelta in funzione del contesto del simbolo in fase di codifica. Essa richiede che più alberi siano mantenuti in questa situazione, che può diventare impegnativa sulla memoria.

L'alternativa è rigenerare ogni albero tutte le volte che è richiesto, ma questo

è lento. Quindi, per la compressione adattiva, è preferibile la codifica aritmetica, poichè la sua velocità è paragonabile a quella della codifica adattiva di Huffman ma richiede meno memoria ed è capace di raggiungere una migliore compressione.
Tuttavia, la codifica di Huffman si rivela essere molto utile per molte applicazioni.

Alcuni commenti sui codici di Huffman

- *Equivalenza di codifica e 20 domande.*
  Facciamo ora una digressione per mostrare l'equivalenza della codifica e del gioco di '20 domande'. Supponiamo che vogliamo trovare la serie più efficiente di domande si-no per determinare un oggetto su una classe. Assumendo che conosciamo la distribuzione di probabilità sugli oggetti, possiamo trovare la sequenza di domande più efficiente? (Per determinare un oggetto, dobbiamo assicurarci che le risposte alla sequenza di domande ne identifica uno in modo univoco dall'insieme degli oggetti possibili; non è necessario che l'ultima domanda abbia come risposta 'si').
  Mostriamo innanzitutto che una sequenza di domande è equivalente ad un codice per l'oggetto. Ogni domanda dipende solo dalle risposte alle domande antecedenti. Poichè la sequenza delle risposte determina in modo univoco l'oggetto, ognuno ha una sequenza diversa di risposte, e se rappresentiamo le risposte si-no rispettivamente da 0 e da 1, abbiamo un codice binario per l'insieme degli oggetti. La lunghezza media di questo codice è il numero medio di domande per lo schema interrogativo.
  Inoltre, dal codice binario, possiamo trovare una sequenza di domande che gli corrisponda, con il numero medio di domande uguale alla lunghezza media della parola di codice. La prima domanda in questo schema è: Nella parola di codice dell'oggetto, il primo bit è uguale a 1? Poichè il codice di Huffman è il migliore per una variabile aleatoria, la

serie ottimale di domande è quella determinata da tale codice.
Nell'Esempio 1.3, la prima domanda ottimale è: X è uguale a 2 o a 3? La sua risposta determina il primo bit del codice di Huffman. Assumendo che la risposta alla prima domanda sia 'si', la successiva domanda dovrebbe essere 'X è uguale a 3?', che determina il secondo bit.
Comunque, non dobbiamo aspettare la risposta alla prima domanda per chiedere la seconda. Possiamo chiedere come nostra seconda domanda 'X è uguale a 1 o a 3?', determinando così il secondo bit del codice di Huffman, indipendentemente dal primo.
Il numero medio di domande EQ in questo schema ottimale soddisfa $H(X) \leq EQ < H(X) + 1$.

- *Codifica di Huffman per codici ponderati*
  L'algoritmo di Huffman per minimizzare $\sum p_i l_i$ può essere applicato ad ogni insieme di numeri $p_i \geq 0$, indipendentemente da $\sum p_i$.
  In questo caso, il codice di Huffman minimizza la somma delle lunghezze del codice ponderato $\sum w_i l_i$ piuttosto che la sua lunghezza media.

- *Codifica di Huffman e 'porzioni' di domande* (codici alfabetici).
  Abbiamo descritto l'equivalenza della codifica con il gioco delle 20 domande. La sequenza ottimale delle domande corrisponde a un codice ottimale per la variabile aleatoria.
  Comunque, i codici di Huffman chiedono domande arbitrarie del tipo '$X \in A$?' per ogni insieme $A \subseteq \{1, 2, ..., m\}$.
  Consideriamo ora tale gioco con un insieme delle domande ristretto. Specificamente, assumiamo che gli elementi di $\chi = \{1, 2, ..., m\}$ siano ordinati in modo che $p_1 \geq p_2 \geq \cdots \geq p_m$ e che la sola domanda consentita sia del tipo '$X > a$?' per qualche a.
  Il codice di Huffman costruito dall'algoritmo potrebbe non corrispondere alle *porzioni* (insiemi del tipo $\{x : x < a\}$). Se prendiamo le lunghezze derivate dal codice di Huffman e le usiamo per assegnare i simboli all'albero portando il primo nodo disponibile al livello corrisponden-

te, costruiremo un altro codice ottimale. Comunque, a differenza dello stesso codice di Huffman, questa è una *porzione di codice*, poichè ogni domanda (ogni bit del codice) divide l'albero in insiemi del tipo $\{x : x > a\}$ e $\{x : x < a\}$.
Illustriamo ciò con un esempio.

Esempio 2.3. Consideriamo l'Esempio 1.3. Il codice costruito dalla procedura della codifica di Huffman non è una porzione di codice. Ma usando le lunghezze del codice dalla procedura di Huffman, vale a dire $\{2, 2, 2, 3, 3\}$, e assegnando i simboli al primo nodo disponibile nell'albero, otteniamo il seguente codice per questa variabile aleatoria:
$1 \rightarrow 00,\ \ 2 \rightarrow 01,\ \ 3 \rightarrow 10\ \ 4 \rightarrow 110,\ \ 5 \rightarrow 111$
Si può verificare che questa è una porzione, conosciuto anche come *codice alfabetico* poichè i codici sono in tale ordine.

- *Codici di Huffman e codici di Shannon*
  Usando le lunghezze del codice di $\lceil log\frac{1}{p_i} \rceil$ (*Codice di Shannon*), per alcuni simboli particolari può essere peggiore del codice ottimale.
  Per esempio, consideriamo due simboli, uno dei quali si verifica con probabilità 0.9999 e l'altro con probabilità 0.0001. Quindi, usando lunghezze pari a $\lceil log\frac{1}{p_i} \rceil$, si ottengono lunghezze del codice di 1 bit e di 14 bit rispettivamente. La lunghezza del codice ottimale è ovviamente 1 bit per entrambi i simboli. Perciò, il codice per il simbolo infrequente è più lungo nel codice di Shannon che in quello ottimale.

  E' vero che le lunghezze per un codice ottimale sono sempre minori di $\lceil log\frac{1}{p_i} \rceil$? Il seguente esempio mostra che questo non è sempre vero.

Esempio 2.4. Consideriamo una variabile aleatoria X con una distribuzione $(\frac{1}{3}, \frac{1}{3}, \frac{1}{4}, \frac{1}{12})$. La procedura della codifica di Huffman si risolve nelle lunghezze del codice di $(2, 2, 2, 2)$ o $(1, 2, 3, 3)$. Entrambi questi codici raggiungono la stessa lunghezza media. Nel secondo codice, il

terzo simbolo ha lunghezza 3, che è più grande di $\lceil log\frac{1}{p_3}\rceil$. Quindi, la lunghezza per un codice di Shannon potrebbe essere inferiore della lunghezza del simbolo corrispondente di un codice ottimale (di Huffman). Questo esempio illustra anche il fatto che l'insieme delle lunghezze per un codice ottimale non è unico.

I codici di Shannon e, in particolare quelli di Huffman, possono essere brevi per simboli individuali. Inoltre, differiscono meno di 1 bit nella lunghezza media del codice (dato che entrambi si trovano tra H e $H+1$).

Proviamo per induzione che il codice binario di Huffman è ottimale.
E' importante ricordare che ci sono molti codici ottimali: invertendo tutti i bit o scambiando due codici della stessa lunghezza avremo un altro codice ottimale. La procedura di Huffman costruisce uno di questi codici ottimali. Per provare l'ottimalità, proviamo innanzitutto alcune proprietà di un particolare codice ottimale. Senza perdita di generalità, assumeremo che le probabilità siano ordinate in modo che $p_1 \geq p_2 \geq \cdots \geq p_m$.
Ricordiamo che un codice è ottimale se $\sum p_i l_i$ è minima.

Lemma 2.1.1. *Per ogni distribuzione, esiste un codice istantaneo ottimale che soddisfa le seguenti proprietà:*

1. *le lunghezze sono ordinate inversamente con le probabilità (se $p_j > p_k$, allora $l_j \leq l_k$).*

2. *i due codici più lunghi hanno la stessa lunghezza.*

3. *due dei codici più lunghi differiscono solo per l'ultimo bit e corrispondono ai due simboli meno probabili.*

I codici che soddisfano queste tre proprietà sono detti canonici.

La chiave per usare i codici canonici in modo efficiente è notare che la codifica di una parola può essere determinata rapidamente dalla lunghezza del suo codice. Per esempio, la parola *said* è il decimo codice di 7 bit. Data questa informazione e quella che il primo codice di 7 bit è 1010100, otteniamo il codice per *said* incrementando 1010100 nove volte, oppure, in modo più efficiente, aggiungendo nove a questa rappresentazione binaria.

I codici canonici entrano in proprio per la decodifica perchè non è necessario memorizzare un albero di decodifica. Tutto ciò che è richiesto è una lista di simboli ordinati secondo l'ordine lessicale dei codici, oltre ad un array che memorizza il primo codice di ogni lunghezza distinta.
Confrontato con l' utilizzo di un albero di decodifica, la codifica canonica è molto diretta.

Teorema 2.1.2. *Il codice di Huffman è ottimale fra tutti i codici univocamente decifrabili; cioè, se C^* è il codice di Huffman e C è un altro qualsiasi codice sulla stessa variabile X, si ha $L(C^*) \leq L(C)$.*

Un limite degli alberi di Huffman è la necessità di utilizzare un numero intero di bit per simbolo. Questo produce una codifica perfetta solo quando tutte le probabilità dei simboli sono potenze negative di due; in caso contrario la differenza tra il numero di bit da utilizzare e quello realmente utilizzato è misura della distanza dall'ottimalità. Questo valore è solitamente molto piccolo quando il numero di simboli è sufficientemente alto ed ha una distribuzione abbastanza uniforme, e al contrario può diventare notevolmente grande quando esiste un simbolo con probabilità di comparire molto maggiore del 50 per cento. Per avvicinarsi maggiormente all'ottimalità si può utilizzare un'altra tecnica detta Codifica Aritmetica.

2.2 La codifica aritmetica

Ideata per la prima volta da Elias come pura teoria matematica, la codifica aritmetica è una tecnica che ha permesso di ottenere compressioni eccellenti usando modelli sofisticati. Il suo obiettivo è quello di codificare ogni simbolo ottimalmente indipendentemente da quanti bit esso richieda, sia esso un numero intero o meno. La sua forza principale è che può codificare arbitrariamente nei pressi dell' entropia. E' stato visto che in media non è possibile codificare meglio dell' entropia (Shannon 1948), quindi, in questo senso, la codifica aritmetica è ottimale.

Per compararla con quella di Huffman, supponiamo che i simboli che devono essere codificati appartengano ad un alfabeto binario, dove hanno probabilità 0.99 e 0.01. L'indice di informazione di un simbolo s con probabilità $Pr[s]$ è $-logPr[s]$ bit, quindi il simbolo con probabilità 0.99 può essere rappresentato usando la codifica aritmetica in poco meno di 0.015 bit. Al contrario, un codificatore di Huffman deve usare almeno 1 bit per simbolo. L'unico modo per prevenire questa situazione con la codifica di Huffman è quello di 'bloccare' diversi simboli contemporaneamente. Un blocco viene trattato come il simbolo che deve essere codificato, così che l'inefficienza per ogni simbolo si estende ora a tutto il blocco, che però è difficile da realizzare perchè dovrebbe esserci un blocco per ogni possibile combinazione di simboli, quindi il numero di blocchi aumenta esponenzialmente con la loro lunghezza; questo effetto viene esasperato se simboli consecutivi provengono da diversi alfabeti.

Il vantaggio della compressione della codifica aritmetica è più evidente in situazioni in cui, ad esempio, un simbolo ha una probabilità particolarmente elevata. Più in generale, l'inefficienza della codifica di Huffman è limitata superiormente da $Pr[s_1]+log\,\frac{2loge}{e} \approx Pr[s_1]+0.086$ bit per simbolo, dove s_1 è il simbolo più probabile. Nell'ultimo esempio descritto sopra, l'inefficienza è quindi almeno di 1.076 bit per simbolo (un limite che può essere banalmente ridotto a 1 bit per simbolo), che è molte volte superiore all'entropia della

distribuzione. D'altra parte, per i testi di inglese compressi, l'entropia è di circa 5 bit per carattere, ed il carattere più comune solitamente è lo spazio, con una probabilità di circa 0.18. Con un tale modello l'inefficienza è minore di $\frac{0.266}{5} \approx 5.3$ per cento. In molte applicazioni di compressione $Pr[s_1]$ è anche più piccolo, e l'inefficienza diventa quasi trascurabile. Inoltre, questo è un limite superiore, ed in pratica l' inefficienza è spesso molto meno di ciò che il limite potrebbe indicare. D'altra parte, quando le immagini vengono compresse, è comune trattare con due alfabeti simbolo e con probabilità altamente asimmetriche, ed in questi casi la codifica aritmetica è indispensabile a meno che l'alfabeto simbolo venga alterato usando una tecnica come quella del blocco.

Un altro vantaggio rispetto alla codifica di Huffman è che quella aritmetica calcola la rappresentazione al volo, è richiesta molta meno memoria elementare per il funzionamento e la riduzione è favorita più facilmente.
Anche i codici canonici di Huffman sono veloci, ma essi sono adatti solo se viene utilizzato un modello statico o semi-statico. La codifica aritmetica è particolarmente adatta come lo è il codificatore per i modelli adattivi ad alte prestazioni, dove si verificano probabilità molto alte, dove molte differenti distribuzioni di probabilità sono memorizzate nel modello, e in cui ogni simbolo è codificato come l' apice di una sequenza di decisioni di livello inferiore.

Uno svantaggio di tale codifica è che è più lenta della codifica di Huffman, specialmente nelle applicazioni statiche o semi-statiche. Inoltre, la natura della produzione implica che non è facile iniziare la codifica nel mezzo di un gruppo compresso. Ciò è in contrasto con la codifica di Huffman, in cui è possibile catalogare i 'punti di partenza' in un testo compresso se il modello è statico. Nel recupero del testo integrale, sono importanti sia la velocità che l'accesso casuale, quindi la codifica aritmetica non può essere appropriata. Per di più, per i tipi di modelli usati per comprimere i sistemi di testo integrale, la codifica di Huffman da una buona compressione come la codifica

aritmetica. Quindi, nelle grandi collezioni di testi ed immagini, è probabile che la codifica di Huffman venga usata per i testi mentre quella aritmetica per le immagini.

Come lavora la codifica aritmetica

La codifica aritmetica può essere piuttosto difficile da comprendere. Sebbene sia interessante, la comprensione non è essenziale. Solo l'interfaccia al programmatore deve essere intesa in modo da utilizzarla.
La produzione di un programmatore aritmetico, come quella di un qualsiasi altro programmatore, è un gruppo di bit. Comunque, è più facile descrivere la codifica aritmetica se prefissiamo il gruppo di bit con 0 e consideriamo la produzione come un numero binario frazionario tra 0 e 1. Nel seguente esempio, il gruppo di produzione è 1010001111, ma verrà trattato come la frazione binaria 0.1010001111. Infatti, per motivi di leggibilità, il numero sarà mostrato come un valore decimale (0.64) piuttosto che come binario, e inizialmente saranno trascurate alcune efficienze possibili.
Comprimiamo, per esempio, la stringa *bccb* dall'alfabeto ternario $\{a, b, c\}$. Useremo un modello adattivo ordine-zero e procederemo con il problema di frequenza zero inizializzando tutti i caratteri di contare fino a 1.
Quando il primo b deve essere codificato, tutti e tre i simboli hanno una probabilità stimata di 1/3. Un programmatore aritmetico memorizza due numeri, *low* e *high*, che rappresenta un sotto intervallo che varia tra 0 ed 1. Inizialmente, $low = 0$ e $high = 1$. L'intervallo tra *low* e *high* è diviso secondo la distribuzione di probabilità che sta per essere codificata. Il passo della codifica aritmetica comporta semplicemente il restringimento dell'intervallo ad uno corrispondente al carattere che deve essere codificato. Quindi, poichè il primo simbolo è b, i nuovi valori sono $low = 0.3333$ e $high = 0.6667$ (lavorando a quattro cifre decimali). Prima di codificare il secondo carattere, c, la distribuzione di probabilità si adatta a causa di b che è già stato visto, così $Pr[a] = 1/4$, $Pr[b] = 2/4$, e $Pr[c] = 1/4$. Il nuovo intervallo è ora suddiviso in queste proporzioni, e la codifica di c implica la modifica dell'in-

tervallo affinchè vada da $low = 0.5834$ a $high = 0.6667$. La codifica continua come mostrato in figura ed alla fine l' intervallo si estende da $low = 0.6390$ a $high = 0.6501$.
A questo punto, il codificatore trasmette il codice mandando ogni valore nell'intervallo da *low* a *high*. Nell'esempio, il valore 0.64 dovrebbe essere adeguato. Il decodificatore simula quello che dovrebbe aver fatto il codificatore. Si comincia con $low = 0$ e $high = 1$ e si divide l'intervallo. Il numero trasmesso, 0.64, si trova nella parte dell'intervallo corrispondente al simbolo b, che deve quindi essere stato il primo simbolo di input. Il decodificatore calcola poi che l'intervallo dovrebbe essere cambiato in $low = 0.3333$ e $high = 0.6667$ e, poichè il primo simbolo è b, la nuova probabilità viene assegnata dove può essere calcolato $Pr[b] = 2/4$. La decodifica procede lungo questa linea fin quando viene ricostruita l'intera stringa.

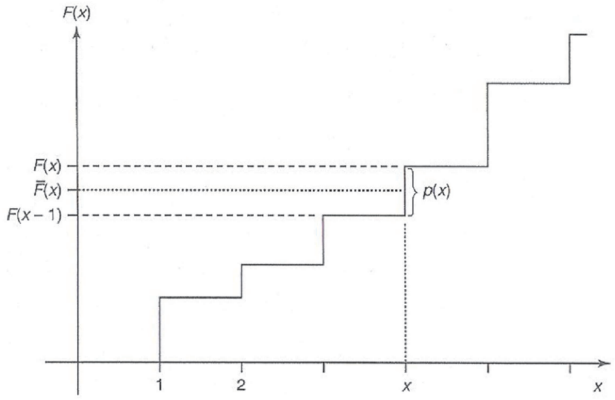
Poichè la codifica aritmetica è in accordo con gli intervalli di probabilità, è normale per un modello fornire *probabilità cumulative* al codificatore ed al decodificatore; questo è il primo passo di ogni algoritmo facile da interpretare. Per la codifica statica e semi-statica, le probabilità cumulative possono essere memorizzate in un array.

2.3 I codici di Shannon - Fano - Elias

Abbiamo visto che le lunghezze del codice $l(x) = \lceil log\frac{1}{p(x)} \rceil$ soddisfano la disuguaglianza di Kraft e possono quindi essere usate per costruire un codice univocamente decifrabile. In questa sezione descriveremo una semplice procedura costruttiva che usa la funzione di distribuzione cumulativa per assegnare i codici.

Senza perdita di generalità, possiamo prendere $\chi = \{1, 2, ..., m\}$, con distribuzione $p(x) > 0$ per ogni x. La funzione di distribuzione cumulativa $F(x)$ è definita come $F(x) = \sum_{a \leq 0} p(a)$.

E' una funzione a gradini e continua a destra, come possiamo notare dalla seguente figura



Consideriamo la funzione cumulativa modificata

$\bar{F}(x) = \sum_{a<x} p(a) + \frac{1}{2}\, p(x) = \frac{F(x-1) + F(x)}{2}$, dove $\bar{F}(x)$ denota la somma delle probabilità di tutti i simboli inferiori ad x maggiorata della metà della probabilità di x. Poichè la variabile aleatoria è discreta, la funzione cumulativa consiste di gradini di altezza $p(x)$. Il valore della funzione $\bar{F}(x)$ è il punto medio del gradino corrispondente ad x.
Dal momento che tutte le probabilità sono positive, $\bar{F}(a) \neq \bar{F}(b)$ se $a \neq b$, e quindi possiamo determinare x se conosciamo $\bar{F}(x)$, semplicemente guardando il grafico della funzione cumulativa troviamo la x corrispondente.
Perciò il valore di $\bar{F}(x)$ può essere usato come un codice per x.
Ma, in generale, $\bar{F}(x)$ è un numero reale esprimibile solo da un numero infinito di bit. Non è quindi efficiente usare il suo esatto valore come un codice per x. Se usiamo un valore approssimato, qual è la precisione richiesta?

Definiamo con $\lfloor \bar{F}(x) \rfloor_{l(x)}$ il *troncamento* di $\bar{F}(x)$ a $l(x)$ bit (cifre binarie).
Usiamo quindi il primo $l(x)$ bit di $\bar{F}(x)$ come un codice per x.

Per definizione di arrotondamento, abbiamo $\bar{F}(x) - \lfloor \bar{F}(x) \rfloor_{l(x)} < \frac{1}{2^{l(x)}}$.
Se $l(x) = \lceil log1p(x) \rceil + 1$, allora $\frac{1}{2^{l(x)}} \leq \frac{p(x)}{2} = \bar{F}(x) - F(x-1)$, e quindi $\lfloor \bar{F}(x) \rfloor_{l(x)}$ si trova all'interno del gradino corrispondente ad x.
Quindi $l(x)$ bit è sufficiente per descrivere x.
La codifica di Shannon-Fano-Elias può anche essere applicata a sequenze di variabili aleatorie. L'idea chiave è di usare la funzione cumulativa della sequenza, espressa con la precisione appropriata, come un suo codice. Una diretta applicazione del metodo ai blocchi di lunghezza n richiederebbe il calcolo delle probabilità e la funzione cumulativa per tutte le sequenze di lunghezza n, un calcolo che crescerebbe esponenzialmente con la lunghezza del blocco. Ma un semplice stratagemma assicura che possiamo calcolare entrambi, la probabilità e la funzione cumulativa della densità sequenzialmente come vediamo ogni simbolo nel blocco, garantendo che il calcolo cresca solo linearmente con la lunghezza del blocco. La diretta applicazione della codifica di Shannon-Fano-Elias potrebbe aver anche bisogno dell'aritmetica, la cui precisione cresce con la grandezza del blocco, che non è pratico quando trattiamo con blocchi lunghi. La codifica aritmetica è, in effetti, un'estensione del metodo di Shannon-Fano-Elias a sequenze di variabili aleatorie che codifica usando precisioni aritmetiche fissate con una complessità che è lineare nella lunghezza della sequenza. Questo metodo sta alla base di molti schemi di pratica compressione come quelli usati nelle compressioni standard come JPEG e FAX.

Abbiamo visto che la codifica di Huffman è ottimale, nel senso che ha lunghezza media minima. Ma ciò che cosa ci dice riguardo le sue prestazioni su ogni sequenza particolare? Cioè, è sempre migliore di ogni altro codice per tutte le sequenze? Ovviamente no, poichè ci sono codici che assegnano parole corte a simboli rari. Tali codici saranno migliori del codice di Huffman in quei simboli. Per formalizzare la questione sull'ottimalità competitiva, consideriamo il seguente gioco: si da una distribuzione di probabilità a due persone e si chiede di designare un codice istantaneo per la distribuzione. Ne

viene quindi estratto un simbolo, e la ricompensa del giocatore A è 1 o -1, a seconda che il codice del giocatore A sia più corto o più lungo di quello del giocatore B. La vincita è zero per i legami.
E' difficile trattare con le lunghezze del codice di Huffman, poichè non c'è un'espressione esplicita di esse. Invece, se consideriamo il codice di Shannon, con le lunghezze $l(x) = \lceil log\frac{1}{p(x)} \rceil$, abbiamo il seguente teorema.

Teorema 2.3.1. *Siano $l(x)$ le lunghezze associate al codice di Shannon, e siano $l'(x)$ le lunghezze associate ad ogni altro codice univocamente decifrabile. Allora $Pr(l(X) \geq l'(X) + c) \leq \frac{1}{2^{c-1}}$. Per esempio, la probabilità che $l'(X)$ è 5 o più bit inferiore a $l(X)$ è minore di $\frac{1}{16}$.*

Dimostrazione. $Pr\ (\ l(X) \geq l'(X) + c) = Pr(\lceil log\frac{1}{p(X)} \rceil \geq l'(X) + c)$

$$\leq Pr(log\ \frac{1}{p(X)} \geq l'(X) + c - 1) = Pr(p(X) \leq 2^{-l'(X)-c+1})$$

$$= \sum_{x:\ p(x)\leq 2^{-l'(x)-c+1}} p(x)\ \leq \sum_{x:\ p(x)\leq 2^{-l'(x)-c+1}} 2^{-l'(x)-(c-1)}$$

$$\leq \sum_{x} 2^{-l'(x)} 2^{-(c-1)}\ \leq 2^{-(c-1)}$$

dato che $\sum 2^{-l'(x)} \leq 1$ per la disuguaglianza di Kraft. □

Quindi, il più delle volte, nessun altro codice può fare meglio di quello di Shannon. Sviluppiamo ora questo risultato.
In un gioco di impostazione teorica, si vorrebbe garantire che $l(x) < l'(x)$ più spesso di quanto $l(x) > l'(x)$.
Il fatto che $l(x) \leq l'(x) + 1$ con probabilità ≥ 12 non lo garantisce.
Mostriamo ora che anche sotto questo criterio più rigoroso, la codifica di Shannon è ottimale. Ricordiamo che la funzione di probabilità $p(x)$ è diadica se $log\ \frac{1}{p(x)}$ è un intero per ogni x.

Teorema 2.3.2. *Per una funzione di probabilità diadica $p(x)$, sia $l(x) = log\ \frac{1}{p(x)}$ la lunghezza del codice binario di Shannon, e sia $l'(x)$ la lunghezza di qualsiasi altro codice binario univocamente decifrabile. Allora*

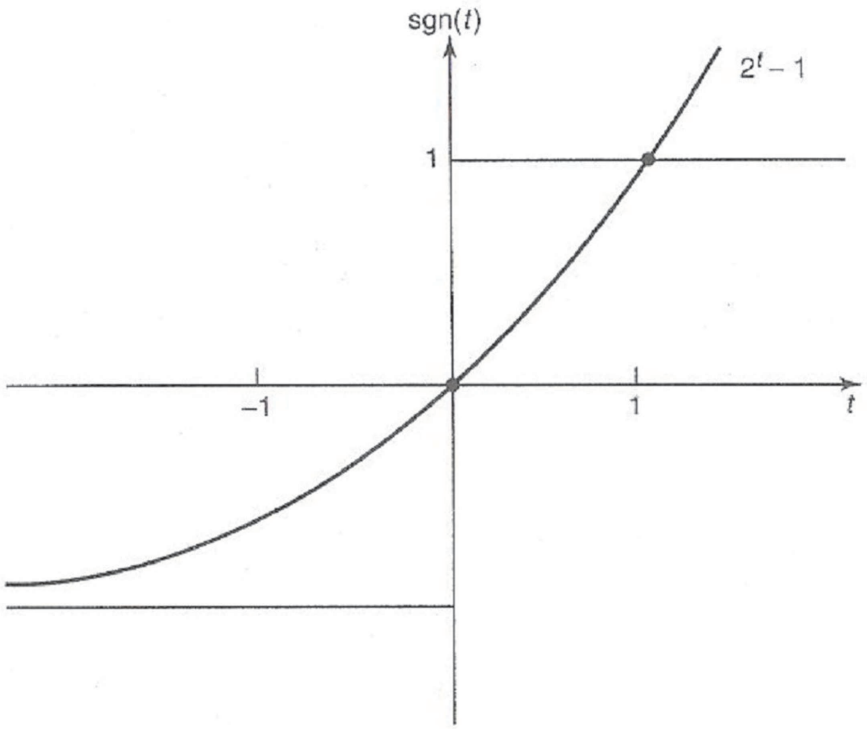
$$Pr(l(X) < l'(X)) \geq Pr(l(X) > l'(X)),$$

vale l'uguaglianza se e solo se $l'(x) = l(x)$ per ogni x.
Quindi, l'assegnazione della lunghezza del codice $l(x) = log\frac{1}{p(x)}$ è ottimale univocamente competitiva.

Dimostrazione. Definiamo la funzione $sgn(t)$ come segue:

$$sgn(t) = \begin{cases} 1 \ \ se \ \ t > 0 \\ 0 \ \ se \ \ t = 0 \\ -1 \ \ se\, t < 0 \end{cases}$$

Dalla seguente figura è facile vedere che $sgn(t) \leq 2^t - 1$ per $t = 0, \pm 1, \pm 2, ...$



Si nota che, sebbene questa disuguaglianza non è soddisfatta per ogni t, lo è per tutti i suoi valori interi. Possiamo quindi scrivere

$$Pr(l'(X)) - Pr(l'(X) > l(X)) = \sum_{x:\, l'(x)<l(x)} p(x) - \sum_{x:\, l'(x)>l(x)} p(x)$$

$$= \sum_x p(x)\ sgn\ (l(x) - l'(x)) = E\ sgn\ (l(X) - l'(X))$$

$$\leq^{(a)} \sum_x p(x)\big(2^{l(x)-l'(x)}-1\big) = \sum_x 2^{-l'(x)}\big(2^{l(x)-l'(x)}-1\big) = \sum_x 2^{-l'(x)} - \sum_x 2^{-l(x)}$$

$$= \sum_x 2^{-l'(x)} - 1 \leq^{(b)} 1 - 1 = 0,$$

dove (a) segue dal limite su $sgn(x)$ e (b) segue dal fatto che $l'(x)$ soddisfa la disuguaglianza di Kraft. Si ha l'uguaglianza se la si ha in (a) e in (b). Nel limite per $sgn(t)$ abbiamo l'uguaglianza solo se t è 0 o 1 [per esempio, $l(x) = l'(x)$ o $l(x) = l'(x)+1$]. L'uguaglianza in (b) implica che $l'(x)$ soddisfa la disuguaglianza di Kraft con l'uguaglianza. Combinando questi due fatti si ha che $l'(x) = l(X)$ per ogni x.

□

Corollario 2.3.3. *Per le funzioni di probabilità non diadiche,*

$$E\ sgn(l(X) - l'(X) - 1) \leq 0,$$

dove $l(x) = \left\lceil log\ \frac{1}{p(x)} \right\rceil$ *e* $l'(x)$ *è un qualsiasi altro codice.*

Abbiamo quindi mostrato che la codifica di Shannon è ottimale sotto vari criteri. In particolare, per p diadico, $E(l - l') \leq 0$, $Esgn(l - l') \leq 0$, e $Ef(l - l') \leq 0$ per ogni funzione f che soddisfa $f(t) \leq 2^t - 1$,
con $t = 0, \pm1, \pm2, ...$

2.4 La generazione di distribuzioni discrete da monete equilibrate

Abbiamo considerato il problema di rappresentare una variabile aleatoria da una sequenza di bit tale che la lunghezza media della rappresentazione sia minima. Si può sostenere che la sequenza codificata sia essenzialmente incomprimibile e quindi che abbia un tasso di entropia vicino ad 1 bit per simbolo. Pertanto, i bit della sequenza codificata sono essenzialmente lanci di una moneta equilibrata.
Facciamo ora una piccola deviazione dalla nostra discussione sulla codifica e consideriamo il problema duale. Quanti lanci di monete ci vogliono per generare una variabile aleatoria X tirata in accordo a una funzione di probabilità specifica $\mathbf{p}$? Consideriamo prima un semplice esempio.

Esempio 2.5. Data una sequenza di lanci di una moneta equilibrata, supponiamo di voler generare una variabile aleatoria X con distribuzione

$$X = \begin{cases} a \text{ con probabilità } 1/2 \\ b \text{ con probabilità } 1/4 \\ c \text{ con probabilità } 1/4 \end{cases}$$

E' facile indovinare la risposta. Se il primo bit è 0, si ha $X = a$. Se i primi due bit sono 10, abbiamo $X = b$. Se troviamo 11, abbiamo $X = c$. E' chiaro che X è la distribuzione desiderata.
Calcoliamo il numero medio di bit richiesti per generare la variabile aleatoria, in questo caso come $\frac{1}{2}(1) + \frac{1}{4}(2) + \frac{1}{4}(2) = 1.5$ bit. Questo è anche l'entropia della distribuzione. Ciò è raro? No.

Il problema generale può essere così formulato.
Abbiamo dato una sequenza di lanci di una moneta equilibrata $Z_1, Z_2, ...$, e vogliamo generare una variabile $X \in \chi = \{1, 2, ..., m\}$ con funzione di probabilità $\mathbf{p} = (p_1, p_2, ..., p_m)$. Lasciamo che la variabile aleatoria T indichi il numero di lanci della moneta nell' algoritmo.

Possiamo descrivere le stringhe di mappatura dell'algoritmo di bit $Z_1, Z_2, ...,$ per i possibili risultati X da un albero binario. Le foglie dell'albero sono contrassegnate dai simboli X di uscita, mentre il cammino per le foglie è dato da una sequenza di bit prodotti dalla moneta.
L'albero che rappresenta l'algoritmo deve soddisfare certe proprietà:

- L'albero deve essere completo (ogni nodo o è una foglia o ha due discendenti nell'albero) e può essere infinito.
- La probabilità di una foglia di profondità k è 2^{-k}. Ogni foglia può essere classificata con lo stesso simbolo di uscita - la probabilità totale di tutte queste foglie dovrebbe essere uguale a quella desiderata del simbolo di uscita.
- Il numero medio di bit ET richiesto per generare X è uguale alla profondità attesa di questo albero.

Ci sono molti algoritmi possibili che generano la stessa distribuzione. Per esempio, la mappa $00 \to a, 01 \to b, 10 \to c, 11 \to a$ anche fornisce la distribuzione $(\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$. Comunque, questo algoritmo usa due bit per generare ogni modello e quindi non è così efficiente come la mappa data prima, che usa solo 1.5 bit per modello. Questo fa sollevare la seguente questione:
Qual è l'algoritmo più efficiente per generare una data distribuzione, e in che modo è connesso all' entropia di tale distribuzione?
Abbiamo bisogno di tanta casualità nei bit quanta ne produciamo nei modelli. Poichè l'entropia è una misura della casualità, ed ogni bit ha un'entropia di 1 bit, ci aspettiamo che il numero di bit usati sarà almeno uguale all'entropia del risultato. Mostreremo questo nel seguente teorema ma, per fare ciò, abbiamo bisogno di un semplice lemma sugli alberi. Lasciamo che Υ denoti l'insieme delle foglie di un albero completo. Consideriamo una distribuzione sulle foglie tale che la probabilità di una di esse alla profondità k nell'albero

è 2^{-k}. Sia Y una variabile aleatoria con questa distribuzione.
Si ha quindi il seguente lemma.

Lemma 2.4.1. *Per ogni albero completo, consideriamo una distribuzione di probabilità sulle foglie tale che la probabilità di una foglia di profondità k è 2^{-k}. Allora la profondità media dell'albero è uguale all'entropia di questa distribuzione.*

Dimostrazione. La profondità media dell'albero è $ET = \sum_{y \in \Upsilon} k(y)\ 2^{-k(y)}$ e l'entropia della distribuzione di Y è

$$H(Y) = -\sum_{y \in \Upsilon} \frac{1}{2^{k(y)}}\ log\ \frac{1}{2^{k(y)}} = \sum_{y \in \Upsilon} k(y)\ 2^{-k(y)},$$

dove $k(y)$ denota la profondità della foglia y. Quindi $H(Y) = ET$.

□

Teorema 2.4.2. *Per ogni algoritmo che genera X, il numero medio di bit usati è maggiore dell'entropia $H(X)$, cioè, $ET \geq H(X)$.*

Dimostrazione. Ogni algoritmo che genera X da bit equi, può essere rappresentato da un completo albero binario. Si classificano tutte le foglie di questo albero per i diversi simboli $y \in \Upsilon = \{1, 2, ...\}$.
Se l'albero è infinito, anche l'alfabeto Υ lo è.
Consideriamo ora la variabile aleatoria Y definita sulle foglie dell'albero, tale che per ogni foglia y di profondità k, la probabilità che $Y = y$ è 2^{-k}.
Dal lemma precedente, la profondità media di questo albero è uguale all'entropia di Y: $ET = H(Y)$. Ora la variabile aleatoria X è una funzione di Y, e quindi abbiamo che $H(X) \leq H(Y)$. Perciò per ogni algoritmo che genera la variabile aleatoria X, abbiamo $H(X) \leq ET$.

□

Teorema 2.4.3. *Se la variabile aleatoria X ha una distribuzione diadica, allora l'algoritmo ottimale per generare X richiede un numero medio di lanci della moneta equilibrata pari all'entropia: $ET = H(X)$.*

Dimostrazione. Il teorema precedente mostra che per generare X abbiamo bisogno almeno di $H(X)$ bit. Per la parte costruttiva, usiamo l'albero di codifica di Huffman per X come albero per generare la variabile aleatoria. Per una distribuzione diadica, la codifica di Huffman è la stessa della codifica di Shannon e raggiunge il limite dell'entropia. Per ogni $x \in \chi$, la profondità della foglia nell'albero di codifica che corrisponde ad x è la lunghezza del codice corrispondente, che è $log\ \frac{1}{p(x)}$. Quindi, quando questo albero di codifica viene usato per generare X, la foglia avrà una probabilità $p(x) = 2^{-log\frac{1}{p(x)}}$. Il numero medio di lanci della moneta è la profondità media dell'albero, che è uguale all'entropia (dato che la distribuzione è diadica).
Perciò, per una distribuzione diadica, l'algoritmo di generazione ottimale raggiunge $ET = H(X)$. □

Cosa succede se la distribuzione non è diadica? In questo caso non possiamo usare la stessa idea, poichè l'albero di codifica per Huffman genera una distribuzione diadica sulle foglie, e no la distribuzione con la quale si inizia. Dato che tutte le foglie dell'albero hanno probabilità del tipo 2^{-k}, segue che potremmo scindere ogni probabilità p_i che non sia di questo tipo in atomi che lo siano. Possiamo poi assegnare questi atomi alle foglie dell'albero. Per esempio, se uno dei risultati x ha probabilità $p(x) = \frac{1}{4}$, abbiamo bisogno di un atomo solo, ma se $p(x) = \frac{7}{8} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8}$, abbiamo bisogno di tre atomi.
Per minimizzare la profondità media dell'albero, dovremmo usare atomi con una probabilità grande il più possibile. Così, data una probabilità p_i, cerchiamo l'atomo più grande del tipo 2^{-k} che sia minore di p_i, ed assegnamo questo atomo all'albero. Calcoliamo poi i restanti e troviamo che l'atomo più grande si inserisce nel resto. Continuando in questo modo, possiamo dividere tutte le probabilità in atomi diadici. Questo processo è equivalente a trovare le estensioni binarie delle probabilità. Supponiamo che l'estensione binaria

della probabilità p_i sia $p_i = \sum_{j \geq 1} p_i^{(j)}$, dove $p_i^{(j)} = 2^{-j}$ o a 0. Allora gli atomi dell'estensione sono i $\left\{ p_i^{(j)} : i = 1, 2, ..., m, j \geq 1 \right\}$.

Dal momento che $\sum_i p_i = 1$, la somma delle probabilità di questi atomi è 1. Assegneremo un atomo di probabilità 2^{-j} ad una foglia di profondità j. Le profondità degli atomi soddisfano la disuguaglianza di Kraft, e quindi dal Teorema 1.2.1, possiamo sempre costruire un tale albero con tutti gli atomi alle giuste profondità.

Teorema 2.4.4. *Il numero medio di bit richiesto dall'algoritmo ottimale per generare una variabile aleatoria X si trova tra $H(X)$ e $H(X) + 2$:*

$$H(X) \leq ET < H(X) + 2$$

.

2.5 Prediction by partial matching -PPM-

A seguito dell'invenzione della compressione aritmetica, uno dei passi più naturali per tentare di aumentare il livello di compressione ottenibile è tentare di considerare una distribuzione di probabilità data da un modello di ordine elevato. La raccolta dei dati necessari per una corretta modellazione sale tuttavia esponenzialmente con il crescere dell'ordine, ed inoltre volendo realizzare un algoritmo che esegue la compressione in un'unica passata sui dati, al momento di codificare i primi simboli spesso non si hanno a disposizione abbastanza informazioni per una buona stima delle probabilità.
Per ovviare a questi problemi, Cleary e Witten [4] hanno proposto, nel 1984, un metodo basato sulle predizioni per corrispondenze parziali (prediction by partial matching, PPM).
La natura adattiva dello schema, e la flessibilità garantita dalla codifica aritmetica, indicano che sarà costruito un modello effettivo di compressione per

ogni file di input che sia ragionevolmente omogeneo. Cleary e Witten hanno riferito che il loro schema era capace di rappresentare testi in Inglese in appena 2.2 bit/carattere. Ciò è molto buono, rispetto ad altri metodi di compressione; per esempio, gli algoritmi di Ziv-Lempel tipicamente richiedono 3.3 bit/carattere [6]. PPM usa modelli a contesto finito. Piuttosto che essere limitato ad una sola lunghezza, esso usa differenti misure a seconda dei contesti osservati nel testo codificato precedentemente - da qui il termine 'partial matching' nel nome.

L'algoritmo tiene quindi conto di più modelli contemporaneamente fino ad un ordine massimo impostato al momento dell'esecuzione. Al momento dell'osservazione di un simbolo, tutte le frequenze vengono aggiornate; questo significa che nel modello di ordine n si deve tener conto che nel contesto degli $n-1$ caratteri precedenti è comparso un determinato carattere. Se a questo punto troviamo un determinato contesto comparso n volte a seguito del quale un determinato simbolo è comparso m volte, potremmo stimare la possibilità di riosservare questo simbolo in un contesto uguale con la quantità n/m.

Al momento di codificare un carattere, l'algoritmo cerca quindi una distribuzione di probabilità per questo simbolo. Come primo tentativo considera il contesto relativo al modello di ordine maggiore. Nel caso in cui il carattere che segue sia già stato visto in tale contesto, questo viene codificato con la probabilità data dal modello in considerazione. Altrimenti viene scritto un carattere speciale, detto *carattere di fuga*, ed il sistema regredisce al modello di ordine minore. Questa operazione se necessario viene ripetuta fino ad arrivare al modello di ordine zero, dove il simbolo viene codificato senza alcuna elaborazione.
Per consentire la codifica, deve essere allocata una determinata probabilità al simbolo di fuga, riducendo così quelle degli altri simboli. E' difficile immaginare una soluzione puramente razionale a questo problema, in quanto si tratta di stimare con quanta facilità si può verificare un evento mai osservato

prima. Per questo, tale problema, detto anche della frequenza zero (zero frequency problem), viene affrontato solitamente in base ad una stima empirica o a risultati sperimentali. Cleary e Witten nel loro primo documento sul PPM dell'84 propongono due metodi:

Chiamato $c(\varphi)$ il numero di occorrenze di un generico carattere φ, $p(\varphi)$ la probabilità allocata per questo evento, ϵ la possibilità del carattere di fuga, C il numero di volte in cui è comparso il contesto considerato e q il numero di caratteri mai comparsi in questo contesto, si può:

- Imporre $p(\varphi) = c(\varphi)/(C+1)$.
  Si verifica che $p(\epsilon) = 1 - \sum(p(\varphi)) = 1/(1+C)$

- Imporre $p(\varphi) = (c(\varphi)-1)/C$.
  Si trova che $p(\epsilon) = 1 - \sum(p(\varphi)) = q/C$

Entrambe queste due proposte si basano sull'idea che la probabilità che in un determinato contesto compaia un simbolo nuovo è tanto più bassa quante più volte quel contesto è stato osservato. La stima più o meno accurata di ϵ incide notevolmente sulla compressione ottenuta dal metodo in quanto, soprattutto all'inizio del messaggio, tale carattere andrà codificato molto spesso.

Numerosi sono stati gli studi tesi a migliorare questa tecnica, originariamente intesa unicamente per la compressione di testo ma in seguito utilizzata anche per dati generici. I principali miglioramenti consistono da un lato in una migliore stima del parametro ϵ, dall'altro in una ottimizzazione delle risorse occupate da questo algoritmo. Si può osservare infatti che la raccolta di dati necessari a raggiungere un modello elevato richiede grandi quantità di memoria e di tempo; per questo le tecniche basate su PPM sono solitamente considerate molto efficaci ma computazionalmente impegnative.

Uno dei modi più efficaci di prevedere i simboli, e quindi di ottenere la compressione, è quello di influenzare le previsioni in base ai più recenti simboli visti. Per esempio, la frase 'Uno dei modi più efficaci di prevedere i simboli, e quindi di ottenere la compression' quasi unicamente predice che il simbolo successivo sia 'e'. Come regola generale, e a condizione che siano note statistiche accurate, più alto è l'ordine del modello, più precise saranno le previsioni, e migliore la compressione. Usando lo stesso esempio, nel contesto più corto 'sion', una serie di altri simboli oltre ad 'e' devono essere considerati come possibili, come ad esempio in 'mansionario'. Sfortunatamente, l'enormità dello spazio campionario per fare previsioni utilizzando lunghi contesti li rende quasi impossibile da gestire per la pratica compressione. Anche restringendo il contesto ai 4 caratteri precedenti (utilizzando i tipici 8 bit bytes) vorrà dire che ci sono 4 miliardi di contesti possibili in eccesso. L'alternativa è di rendere lo schema adattivo. I modelli adattivi di ordine inferiore sono veloci a stabilire statistiche utili ma, nel corso di un testo lungo, raggiungeranno solo compressioni limitate.

Questo dilemma viene risolto nello schema del PPM utilizzando un modello adattivo basato su un contesto di lunghezza variabile. Come già detto, ad ogni passo di codifica, il contesto più lungo precedentemente incontrato viene utilizzato per prevedere il prossimo carattere. Se il simbolo è nuovo per questo contesto, viene trasmesso un codice di fuga e il contesto è abbreviato lasciando cadere un simbolo. Questo processo della trasmissione di un codice di fuga e dell'accorciamento del contesto continuerà finchè il simbolo non sia trasmesso con successo. Se il simbolo corrente è nuovo anche per il contesto di ordine zero, allora verrà trasmessa una fuga finale, ed il simbolo sarà codificato come un codice di 8 bit. Il modello adattivo potrà poi aggiungere il simbolo corrente a tutti i contesti applicabili.

L'idea di base del PPM è, quindi, di evitare il problema di frequenza zero passando ad un contesto di ordine inferiore.

Supponiamo che stiamo codificando una sequenza $X_1, X_2, ..., X_n$ e vogliamo stimare le probabilità del simbolo successivo, detto X_{j+k}, dati i k simboli precedenti. Se alcuni simboli non si sono mai verificati prima in questo contesto, allora abbiamo il problema di frequenza zero. Non sappiamo come stimare le probabilità di questi simboli. Quindi, grazie al PPM, il codificatore dice al decodificatore che il simbolo X_{j+k} che viene dopo non si è mai verificato, e che dovrebbero (entrambi) passare al contesto dei precedenti $k-1$ simboli, invece che a quello dei k simboli. Alcuni dettagli:

- Piuttosto che usare un modello di Markov di ordine k, se ne usano diversi, cioè modelli di ordine $0, 1, ..., k_{max}$, dove k_{max} è il modello di ordine più grande considerato. Come il codificatore codifica la sequenza, esso opera per ciascuno di questi modelli. Questo permette di calcolare funzioni di probabilità condizionata:

  p (simbolo successivo | k_{max} simboli precedenti)
  p (simbolo successivo | $k_{max} - 1$ simboli precedenti)
  $\vdots$
  p (simbolo successivo | simbolo precedente)
  p (simbolo successivo)

- Si aggiunge un altro simbolo all'alfabeto, $A_{N+1} = \epsilon$ (da 'escape', fuga). Questo simbolo indica che il codificatore sta passando dall' ordine k all'ordine $k-1$.

- Per scegliere le probabilità del simbolo successivo, usiamo l'ordine più alto del modello k tale che i k simboli precedenti seguiti dal simbolo successivo si sono verificati almeno una volta in passato. Ossia, il codificatore mantiene la fuga finchè non trova un tale ordine.

- Cosa accade se un simbolo non si è mai verificato prima in nessun contesto? Ovvero, cosa accade se è la prima volta che si verifica un simbolo?

In questo caso, il codificatore usa un modello di ordine '$k = -1$' per codificare il simbolo. (Chiamare l'ordine '-1' è insignificante, certo. Ciò significa che stiamo diminuendo l'ordine da $k = 0$, siamo cioè fuggiti da $k = 0$.)

Esempio 2.6. : 'abracadabra' con $k_{max} = 2$

I simboli che vengono codificati sono:
$\epsilon, \epsilon, \epsilon, a, \epsilon, \epsilon, \epsilon, b, \epsilon, \epsilon, \epsilon, r, \epsilon, \epsilon, a, \epsilon, \epsilon, \epsilon, c, \epsilon, \epsilon, a, \epsilon, \epsilon, \epsilon, d, \epsilon, \epsilon, a, \epsilon, b, r, a$
e gli ordini corrispondenti:
$2, 1, 0, -1, 2, 1, 0, -1, 2, 1, 0, -1, 2, 1, 0, 2, 1, 0, -1, 2, 1, 0, 2, 1, 0, -1, 2, 1, 0, 2, 1, 2, 2$

Ci si chiede come ciò possa raggiungere la compressione.
Il numero di simboli (incluso ϵ) è molto più grande del numero originale. Da un lato, si potrebbe pensare che ciò richiede più bit! Dall'altro, si noti che molte di queste 'fughe' si verificano con probabilità 1 e quindi non è necessario nessun bit per essere inviati.

Esempio 2.7. Consideriamo l'alfabeto $\{a, b\}$ e sia $k_{max} = 1$.
Supponiamo che la stringa che vogliamo codificare inizi con *aabb*...
Inviamo i seguenti simboli: $\epsilon, \epsilon, a, \epsilon, a, \epsilon, \epsilon, b, \epsilon, b, ...$
Considerimo ora, per ognuno di questi simboli, le probabilità per a, b, ϵ.
Prendiamo in considerazione solo i casi in cui la probabilità è 0 o 1.
Solo i simboli che sono già stati visti prima nel contesto attuale (i k simboli precedenti) possono essere codificati in questo contesto. Se A_i non si è verificato, assegnamo la sua probabilità a 0 e quindi codifichiamo ϵ. La ϵ dice al decodificatore che il simbolo che viene dopo è uno di quelli che non è stato visto prima nel contesto attuale. Cosa significa mandare ϵ a $k = -1$? Ciò potrebbe essere usato per indicare la fine della sequenza

Può sembrare che le performance del PPM dovrebbero sempre migliorare quando viene aumentata la lunghezza massima del contesto, ottenendo

quindi la migliore compressione. Questo comportamento è abbastanza tipico. La ragione è che, sebbene i contesti più lunghi forniscano previsioni più specifiche, offrono anche una possibilità molto maggiore di non dare luogo ad alcuna previsione. Questo fa sì che il meccanismo di fuga sia utilizzato più frequentemente per ridurre la lunghezza del contesto fino al punto in cui iniziano ad apparire le previsioni. Ed ogni operazione di fuga porta ad una piccola penalità sull'efficienza della codifica.

Abbiamo quindi visto che il PPM può essere applicato ad un gruppo di caratteri, ma questa tecnica può anche essere utilizzata per codificare gruppi in cui l'alfabeto non è così ristretto. In particolare, siamo interessati a codificare gruppi di numeri interi rappresentanti parole. Ogni intero del gruppo sarà compreso nell'intervallo da 1 a $n+1$, dove n è il massimo intero apparso finora. Un tale gruppo è generato dalla mappatura di distinte parole su numeri interi assegnati in modo sequenziale, quindi, per esempio, 'to be or not to be' è rappresentato da '1 2 3 4 1 2 '.
Questa è quindi un'alternativa al procedimento del PPM di imporre un fissato massimo limite superiore universale sulla lunghezza del contesto, che in tal modo può variare in base alla codifica. E' possibile memorizzare il modello in un modo che dà accesso rapido alle previsioni di ogni contesto, eliminando la necessità di imporre un limite arbitrario. PPM* [5] è il nome di questo approccio, in cui non c'è un limite *a priori* sulla lunghezza del contesto.

Bibliografia

[1] Thomas M. Cover, Joy A. Thomas, '*Elements of information theory*', 2006, Wiley Interscience.

[2] Jeffrey R. Sampson, '*Adaptive information processing: an introductory survey*', 1976, Springer-Verlag.

[3] Ian H. Witten, Alistair Moffat, Timothy C. Bell., '*Managing gigabytes: compressing and indexing documents and images*', 1999, Morgan Kaufmann.

[4] John G.Cleary, Ian H.Witten, '*Data compression using adaptive coding and partial string matching*', 1984, IEEE Transactions on Communications.

[5] John G.Cleary, W.J.Teahan, Ian H.Witten, '*Unbounded length contexts for PPM*', Department of Computer Science, University of Waikato, New Zealand.

[6] Alistair Moffat, '*Implementing the PPM Data Compression Scheme*', 1990, IEEE.