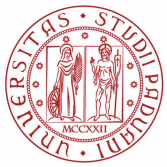

Università degli Studi di Padova

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria dell'Informazione

Compressione dati: uno studio sulla proprietà di equipartizione asintotica

Relatore: Prof. Giancarlo Calvagno

Laureando: *Edoardo Furlan*

Anno Accademico 2023/2024
Data di Laurea: 26-09-2024

Sommario

In questa tesi si andrà ad introdurre il concetto di compressione dei dati, ponendo l'attenzione sulle basi, derivanti dalla Teoria dell'Informazione, che ne permettono una solida definizione matematica. In particolare, si introdurrà il concetto di entropia di un messaggio e come esso sia strettamente collegato al funzionamento degli algoritmi di compressione. Il contenuto principale sarà lo studio della cosiddetta *Asymptotic Equipartition Property*, o "Proprietà di Equipartizione Asintotica", che mostra come si comporta asintoticamente (cioè al crescere della sua lunghezza) un messaggio aleatorio. Vedremo come questa proprietà sia utile nel fornire una certa prevedibilità riguardo ai messaggi da codificare. Infine, si introdurranno dei codici scritti in linguaggio di programmazione Python, che andranno ad implementare alcuni degli algoritmi visti nel corso del documento.

Indice

Notazioni

Le seguenti notazioni, fondamentali per i concetti di probabilità richiesti dalle varie trattazioni matematiche, saranno utilizzate in questo documento:

- X: variabile aleatoria.
- $\mathcal{X}$: alfabeto della v.a. X.
- $p(x)$: probabilità del simbolo x.
- $F(x)$: funzione di distribuzione della v.a. X. Definita, per una variabile discreta, come: $F(x) = \sum_{x \leq a} p(a)$ dove $p(x)$ è la densità di probabilità discreta della variabile aleatoria.

  Per una variabile continua, la funzione viene definita come: $F(x) = \int_{-\infty}^{x} f(x)dx$ dove $f(x)$ è la densità di probabilità continua della variabile aleatoria.
- $x^n = x_1, x_2, \ldots, x_n$.
- $\mathbf{x}$: indica un vettore.

Capitolo 1

Compressione dei dati

1.1 Introduzione

La compressione dei dati è una tecnica che pone come obiettivo la rappresentazione dell'informazione nel modo più compatto possibile, occupando quindi meno "spazio" possibile. Se parliamo di un file salvato in memoria, vogliamo che esso richieda pochi byte per essere immagazzinato.

Se invece parliamo di un messaggio inviato attraverso un mezzo di telecomunicazione vogliamo che esso richieda, ad esempio, poca banda, per diminuire costi ed eventuale perdita d'informazione. Senza lo sviluppo di algoritmi di compressione efficienti, non sarebbe fisicamente sostenibile il sempre maggior traffico dati sull'Internet e nemmeno la loro archiviazione in memoria fisica.

Esistono due grandi categorie in cui vengono suddivisi questi algoritmi: "Senza perdita" (*Lossless*) e "Con perdita" (*Lossy*). I primi garantiscono una perfetta ricostruzione del messaggio dopo la decompressione, e sono necessari per applicazioni come la compressione di testi o la compressione di immagini mediche, per cui tutti i dettagli devono necessariamente essere conservati; mentre i secondi comportano una parziale perdita di informazione. Ad esempio, una registrazione audio o video generalmente non richiede un'elevata fedeltà nella ricostruzione ed accetta delle perdite. Ovviamente gli algoritmi *Lossy* sono associati ad un rapporto di compressione (cioè il rapporto tra dimensione del file compresso e dimensione del file originale) più piccolo rispetto ai primi, garantendo quindi una maggiore efficienza della compressione.

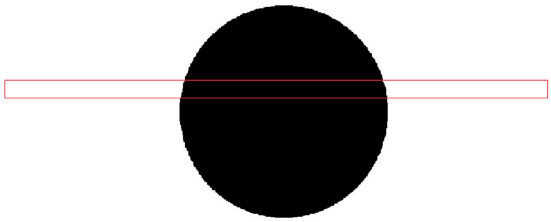

Figura 1.1: Semplice immagine bianco e nero, la riga evidenziata in rosso è quella che stiamo considerando per la codifica RLE.

1.2 Esempio: RLE (Run length encoding)

Partiamo subito con un esempio molto semplice.

In un'immagine in bianco e nero solo due bit sono richiesti per il colore, '1' per il nero e '0' per il bianco. Se consideriamo la riga evidenziata in rosso della Figura 1.1, otteniamo questa particolare sequenza (ovviamente molto semplificata, e solo indicativa per questo esempio):

[0,0,0,0,0,1,1,1,1,1,0,0,0,0,0]

In questo esempio possiamo dire che i bit sono estremamente *ridondanti*, cioè la loro distribuzione è molto semplice, dato che essi sono molto raggruppati ('zeri' vicini agli 'zeri' ed 'uni' vicini agli 'uni'. Per questo motivo, possiamo utilizzare una notazione alternativa:

(0,5),(1,5),(0,5)

Questa è la cosiddetta *Run Length Encoding*: consiste nello specificare un certo carattere, seguito dal numero di volte, in sequenza, in cui questo carattere si ripete. In questo modo, la codifica ci permette di ricostruire l'immagine grazie alle "istruzioni" che la nuova rappresentazione fornisce: dobbiamo utilizzare 5 '0', seguiti da 5 '1' e di nuovo da 5 '0'.

Capitolo 2

Codifica entropica

In compressione dati esistono degli algoritmi che permettono in modo automatico di comprimere un file: da un algoritmo molto semplice come quello visto nell'esempio di prima ai più famosi ed utilizzati "zip" o "jpeg".

La famiglia di algoritmi che andremo a considerare è definibile come "codifica entropica".

Questa tecnica si basa sull'idea che ai simboli (a lunghezza fissa) più *frequenti* che compongono il messaggio originale, debbano essere assegnati i simboli (a lunghezza variabile), appartenenti all'alfabeto di compressione, con lunghezza minore. L'opposto deve invece avvenire per i simboli *meno frequenti*. In questo modo, nel messaggio compresso saranno presenti con maggiore frequenza i simboli più brevi: questo comporta una minore lunghezza *media* dei messaggi. In termini probabilistici, diciamo che diminuisce il valore atteso dell'intera codifica.

Questa idea di codificare simboli più probabili in simboli più corti è però una caratteristica comune nella trasmissione dell'informazione, in quanto rispecchia la necessità di utilizzare la minor quantità di risorse possibili all'interno di un sistema di telecomunicazioni: per fare un esempio, anche nel famoso Codice Morse si fa uso di questa tecnica, in quanto lettere più frequenti nell'alfabeto inglese sono associate a simboli più corti: la E è associata al singolo punto, la T alla singola linea.

L'altra importante idea di questi algoritmi sta quindi nel termine entropica. L'entropia (definita secondo Shannon), è un elemento caratteristico di ogni messaggio, e fornisce sia un limite all'efficienza del codificatore, sia, di conseguenza, una indicazione della quantità di simboli necessari a codificare il messaggio. Vedremo ora più in dettaglio questo concetto fondamentale.

2.1 Entropia

Seppur i due concetti siano collegati, l'entropia che andremo a definire in questo documento, cioè dal punto di vista della Teoria dell'Informazione, è diversa dalla classica concezione derivante dalla seconda legge della termodinamica. Anche la prima infatti si può dire che fornisca una misura di quanto sia disordinato un sistema. Essa infatti aumenta quando un certo messaggio, che d'ora in poi modelleremo sempre come una variabile aleatoria X, presenta molta incertezza.

Introduciamo prima il concetto di **informazione** di una variabile aleatoria:

Definizione 1. *Data una variabile aleatoria X, con alfabeto $\mathcal{X}$ e distribuzione di probabilità $p(X)$, la funzione di informazione del simbolo $a \in \mathcal{X}$:*

$$i_X(a) = -\log_2 p_X(a) \tag{2.1}$$

Questa proprietà definisce quanta informazione fornisce il verificarsi di un certo evento a: più improbabile l'evento e maggiore l'informazione che esso porta.

L'**entropia** della variabile aleatoria è invece definita come il valore atteso dell'informazione:

$$H(X) = -\sum_{a \in \mathcal{X}} p(a) log_2 p(a) \tag{2.2}$$

Essa è misurata in **bit** ed è una misura della **incertezza media** della variabile aleatoria.

Prendiamo ad esempio una v.a. **uniforme** con cardinalità dell'alfabeto uguale a 32. Dalla (2.2), otteniamo $H(X) = 5$ Bit: questi sono i bit richiesti per rappresentarne ogni elemento. L'uniformità è la condizione di massima incertezza per una variabile aleatoria perché sappiamo, a priori, cioè conoscendo la sua distribuzione di probabilità, che nessuno degli eventi possibili ha più probabilità di un altro di verificarsi.

Se invece consideriamo una variabile aleatoria la cui probabilità è distribuita solo su un certo numero di elementi (al limite una variabile aleatoria degenere), cioè alcuni di essi hanno associata una probabilità molto grande rispetto ad altri, l'entropia va a diminuire poiché, intuitivamente, l'incertezza media della variabile aleatoria è diminuita, cioè si ha più **certezza** riguardo a quali simboli appariranno.

2.1.1 Entropia relativa

Tra le molte proprietà dell'entropia, per la nostra trattazione ci concentreremo su una in particolare.

Definizione 2. *L'entropia relativa tra due masse di probabilità p(x) e q(x) è definita come*

$$D(p||q) = \sum_{x \in X} p(x) log \frac{p(x)}{q(x)} \tag{2.3}$$

$$= E\left[log \frac{p(X)}{q(X)}\right] \tag{2.4}$$

Come per l'entropia, la misuriamo in bit.

Possiamo pensare ad essa come ad una "distanza" (seppur non nel senso stretto del termine) tra le due distribuzioni.

Se due variabili aleatorie con distribuzioni di probabilità p e q sono identiche (stesso alfabeto e stesse probabilità per ogni evento), la loro entropia relativa è nulla, poiché le due masse di probabilità hanno distanza zero. Possiamo quindi dire che, più le due distribuzioni di probabilità sono vicine, cioè minore è la loro entropia relativa, e più è efficiente "assumere che la distribuzione considerata sia q quando la vera distribuzione è p".

2.2 Entropia ed efficacia della compressione

2.2.1 Codici istantanei e disuguaglianza di Kraft

Chiariremo ora come il concetto di entropia del messaggio sia strettamente legato all'efficacia degli algoritmi citati.

Quando ci riferiamo al termine codice, stiamo considerando l'insieme di simboli che vengono utilizzati per rappresentare il messaggio compresso.

Iniziamo introducendo alcune definizioni:

Definizione 3. *Un codice si dice istantaneo se nessun suo simbolo è prefisso di un altro simbolo.*

Questo permette una immediata (o, appunto, istantanea) decodifica, poiché durante la lettura di un messaggio codificato un qualsiasi simbolo x_i, appartenente

all'alfabeto del codice, viene riconosciuto non appena si arriva alla fine della lettura dei bit che gli corrispondono, senza dover aspettare ulteriori informazioni.

Ad esempio, è istantaneo il codice che utilizzi i simboli: 0, 10, 110, 111. Notate come nessuna di queste quattro sequenze può essere prefisso per un altro simbolo.

Non è istantaneo il codice: 10, 00, 11, 110, poiché, ad esempio, se durante la lettura incontriamo la sequenza '11', dovremo ancora aspettare il prossimo bit per essere sicuri di non esserci imbattuti nella sequenza '110'.

Supponendo di voler utilizzare un codice istantaneo, il seguente teorema fornisce un limite che la lunghezza dei simboli devono rispettare.

Nota: la dimostrazione può essere estesa ad un più generale alfabeto D-ario ma per coerenza continuiamo con un alfabeto binario.

Teorema 1. *(Disuguaglianza di Kraft) Per ogni codifica istantanea di alfabeto binario, le lunghezze dei simboli, definite come* $l_1, l_2, \ldots, l_m$ *devono soddisfare la diseguaglianza*

$$\sum_i 2^{-l_i} \leq 1 \tag{2.5}$$

Dimostrazione: *Consideriamo un albero binario, cioè nel quale ogni nodo ha 2 figli. Allora, ogni simbolo è rappresentato da una foglia dell'albero (per un codice istantaneo, poiché nessun simbolo può essere "antenato" di un altro simbolo). Sia* l_{max} *la lunghezza del simbolo più lungo, consideriamo tutti i nodi a quel livello. Alcuni sono simboli della codifica, mentre altri sono discendenti di simboli (ovviamente questi sono nodi "non considerati" per la codifica, in modo da permettere l'istantaneità). Un simbolo a livello i avrà esattamente* $2^{l_{max}-l_i}$ *discendenti al livello* l_{max}*. Ognuno di questi nodi è disgiunto ed in numero minore ad* $2^{l_{max}}$*. Sommando tutti i simboli otteniamo*

$$\sum 2^{l_{max}-l_i} \leq 2^{l_{max}-l_i} \tag{2.6}$$

che è equivalente a dire

$$\sum_i 2^{-l_i} \leq 1 \tag{2.7}$$

che è esattamente (2.5).

2.2.2 Codifica ottima

Il problema consiste nel trovare un codice istantaneo la cui lunghezza abbia il minor valore atteso possibile. Dobbiamo perciò *minimizzare* la funzione:

$$L = \sum p_i l_i \tag{2.8}$$

su tutti gli interi $l_1, l_2, \ldots, l_m$ che soddisfano la disuguaglianza (2.5).

Attraverso un algoritmo di minimizzazione, e trascurando il vincolo di interezza delle lunghezze (numeri decimali possono poi essere arrotondati all'intero più vicino), troviamo che l'insieme di lunghezze migliore è:

$$l_i^* = -log_2 p_i \tag{2.9}$$

Perciò, il valore atteso della lunghezza diventa

$$L^* = \sum p_i l_i^* = -\sum p_i log_2 p_i = H(X) \tag{2.10}$$

Poiché, appunto, il set richiesto è di interi, ci possiamo limitare a considerare l^* come un minimo ottimale, e cercare l'insieme di interi l_i più vicino.

Enunciamo ora un teorema fondamentale, che verifica l'ottimalità dell'insieme di lunghezze cercato:

Teorema 2. *Il valore atteso L della lunghezza di un codice istantaneo binario per una variabile aleatoria X è sempre maggiore o uguale all'entropia $H(X)$, ovvero:*

$$L \geq H(X) \tag{2.11}$$

Dimostrazione: *Possiamo scrivere la differenza tra il valore atteso e l'entropia come:*

$$L - H(X) = \sum p_i l_i - \sum p_i log_2 \frac{1}{p_i} \tag{2.12}$$

$$= -\sum p_i log_2 2^{-l_i} + \sum p_i log_2 p_i \tag{2.13}$$

Ora, sia $r_i = 2^{-l_i} / \sum_j 2^{-l_j}$ *e* $c = \sum 2^{-l_i}$*, otteniamo*

$$L - H = \sum p_i log_2 \frac{p_i}{r_i} - log_2 c \tag{2.14}$$

$$D(p||r) + log_2 \frac{1}{c} \geq 0 \tag{2.15}$$

Derivante dal fatto che l'entropia relativa è non-negativa e che per la disuguaglianza di Kraft $c \leq 1$. *Questo conclude la dimostrazione.*

La trattazione matematica che abbiamo appena fatto comporta un risultato molto importante.

Abbiamo dimostrato come l'entropia sia *limite inferiore alla lunghezza media di qualsiasi algoritmo di compressione*. Questo vuol dire che, conoscendo l'entropia della variabile aleatoria secondo la quale sono distribuiti i simboli del messaggio da inviare, potremo dare una chiara misura della sua efficienza, misurando l'entropia del messaggio codificato.

In sostanza, più il numero medio di simboli dopo la codifica si avvicina all'entropia, e più efficiente è l'algoritmo.

2.3 Teorema di Shannon per codifica a blocchi

Introduciamo ora un teorema fondamentale per la compressione dati.

Il teorema di Shannon per la codifica a blocchi collega il limite inferiore alla compressione dati, che abbiamo appena dimostrato, al limite superiore sotto il quale possiamo essere certi stia il valore atteso della lunghezza dei simboli generati da un algoritmo di codifica ottimo.

Considerando simboli sorgente X^n il teorema afferma che

Teorema 3. *Per* ***ogni*** *codice binario decodificabile* C, *la lunghezza di simbolo media* L *soddisfa*

$$L \geq H(X^n). \tag{2.16}$$

Inoltre ***esiste*** *un codice prefix-free tale per cui*

$$L < H(X^n) + 1 \tag{2.17}$$

Dimostrazione: *La prima parte del teorema è già stata dimostrata. Per la seconda, consideriamo un codice che abbia dizionario* $D = \{a_1, a_2, \ldots, a_n\}$ *e lunghezze dei simboli* $l_i = \lceil log_{\frac{1}{2}} p(a_i) \rceil$. *Detta ora* μ_s *la mappa di codifica, otteniamo*

$$L(\mu_s(a_i)) = l_i < log_{\frac{1}{2}} p(a_i) + 1 \tag{2.18}$$

e prendendo il valore atteso da entrambi i lati

$$L < E[i(x^n)] + 1 = H(X^n) + 1 \tag{2.19}$$

Se consideriamo una sorgente di bit indipendenti, possiamo riscrivere le disequazioni come

$$nH(X) \leq L < nH(X) + 1 \tag{2.20}$$

e, dividendo tutto per la lunghezza di simbolo n

$$H(X) \leq \frac{1}{n}L < H(X) + \frac{1}{n}. \tag{2.21}$$

Da questo teorema possiamo anticipare un risultato che vedremo meglio nel prossimo capitolo. Notiamo infatti un beneficio nell'aumento della lunghezza dei simboli. In questo caso infatti più grande scegliamo n e più il teorema garantisce che sia possibile trovare una codifica con simboli di lunghezza media vicini all'entropia.

2.4 Esempi

2.4.1 Codifica di Huffman

Consideriamo una sequenza di N variabili aleatorie indipendenti ed identicamente distribuite X con alfabeto $\mathcal{X} = \{0,1\}$ e distribuzione di probabilità $p(0) = 0.9$ e $p(1) = 0.1$.

La sequenza da codificare viene suddivisa in simboli di lunghezza 3, perciò con alfabeto $\mathcal{X}^3 = [000, 001, 010, 011, 100, 101, 110, 111]$. Calcoliamo la distribuzione di probabilità dei simboli: essendo le variabili X i.i.d. otteniamo: $p(000) = p(0) * p(0) * p(0) = 0.729 \quad p(001) = p(010) = p(100) = 0.081 \quad p(011) = p(110) = p(101) = 0.009 \quad p(111) = 0.001$.

L'algoritmo di Huffman prevede 2 principali passaggi, ad ogni iterazione:

1. I due simboli con probabilità minore vengono aggiunti come figli ad un albero (binario in questo esempio, in quanto la codifica che effettuiamo è binaria) ed eliminati dalla lista dei simboli disponibili

2. Al loro nodo genitore viene associata la somma delle due probabilità, ed esso viene aggiunto alla lista dei simboli

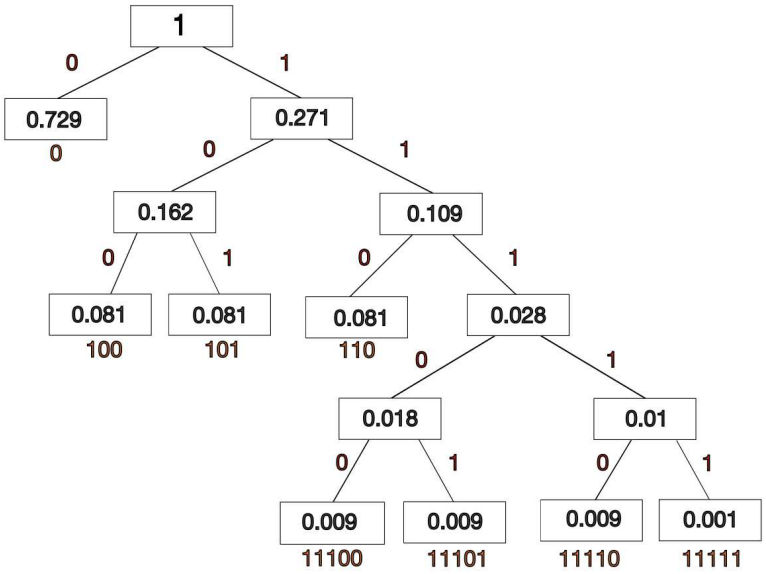

Figura 2.1: Albero di Huffman associato all'esempio.

A questo punto viene percorso l'albero dalla radice fino ad ogni foglia, in modo che ad ogni simbolo venga associata una stringa binaria. In Figura 2.1 è mostrata la codifica dei simboli considerati. Nelle foglie è scritta la probabilità delle sequenze da codificare mentre nei nodi la somma delle probabilità dei rispettivi sotto-alberi, fino a raggiungere la probabilità totale nella radice. Nei rami si vede invece lo sviluppo della codifica: 0 per ogni ramo sinistro ed 1 per ogni ramo destro. Infine, sotto ad ogni foglia è specificata la codifica associata ad ogni simbolo: $000 \to 0, 001 \to 100, 010 \to 101, \ldots$

Notiamo che ai simboli *più probabili* viene assegnato un codice più corto, portando ad una diminuzione della lunghezza media. Se infatti il valore atteso della lunghezza nel codice originale è 3 (banalmente, tutti i simboli hanno la stessa lunghezza), per il messaggio codificato vale:

$$L = \sum_{x \in \mathcal{X}^3} p(x) l(x) = 1.6 \tag{2.22}$$

dove $l(x)$ è la lunghezza del simbolo di codice associato al simbolo x e l'unità di misura rimane il *bit*.

2.4.2 Codifica Shannon-Fano-Elias

Data una v.a. X con alfabeto $\mathcal{X} = \{1, 2, \ldots, n\}$ e funzione di distribuzione $F(x)$, consideriamo:

$$\bar{F}(x) = \sum_{x<a} p(a) + \frac{1}{2}p(x) \tag{2.23}$$

Poiché la variabile aleatoria è discreta, la funzione di distribuzione è a gradini, ed il valore di $\bar{F}(x)$ corrispondente ad x rappresenta il valore di mezzo del gradino. Dato che la conoscenza di $\bar{F}(x)$ fornisce una diretta conoscenza di x, possiamo usarne i valori per creare un codice. Utilizzando

$$l(x) = \left\lceil log_2 \frac{1}{p(x)} \right\rceil + 1 \tag{2.24}$$

bit per il simbolo x (formula per cui i simboli più probabili ottengono rappresentazioni più corte), prendiamo come esempio la v.a. con $\mathcal{X} = \{1, 2, 3, 4\}$, $p(1) = 0.25$, $p(2) = 0.5$, $p(3) = 0.125$, $p(4) = 0.125$ e con $\bar{F}(1) = 0.125, \bar{F}(2) = 0.5, \bar{F}(3) = 0.8125, \bar{F}(4) = 0.9375$. A questo punto possiamo convertire i valori di $\bar{F}(x)$ in binario ed utilizzare un numero di cifre corrispondente alla lunghezza data dall'equazione (2.24) per ottenere la codifica $1 \to 001$, $2 \to 10$, $3 \to 1101$, $4 \to 1111$.

Questa codifica mostra però delle imperfezioni: ad esempio, notiamo che l'ultimo bit degli ultimi due simboli potrebbe essere rimosso. Dalle idee presentate in questo paragrafo, però, possiamo introdurre una migliore codifica.

2.4.3 Codifica aritmetica

Partiamo sempre dall'idea di utilizzare un codice basato sul valore intermedio derivante dalla funzione a gradini di una certa densità di probabilità $p(x^n)$. Assumiamo, sempre per continuità, che l'alfabeto originale sia binario.

Per ordinare le stringhe di bit, diremo che x è maggiore di y se $x_i = 1$ e $y_i = 0$ per il primo i tale che $x_i \neq y_i$. Organizziamo le stringhe come foglie di un albero di profondità n, dove ogni livello corrisponde ad un bit.

Ora, il valore della funzione di distribuzione di $F(x^n)$ è dato dalla somma delle probabilità dei sotto-alberi a sinistra di x^n. Se quindi utilizziamo $\bar{F}(x)$per generare la codifica, possiamo calcolarne il valore tramite un calcolo sequenziale di $F(x)$ e $p(x)$: cioè, ad ogni iterazione dell'algoritmo calcoliamo la probabilità

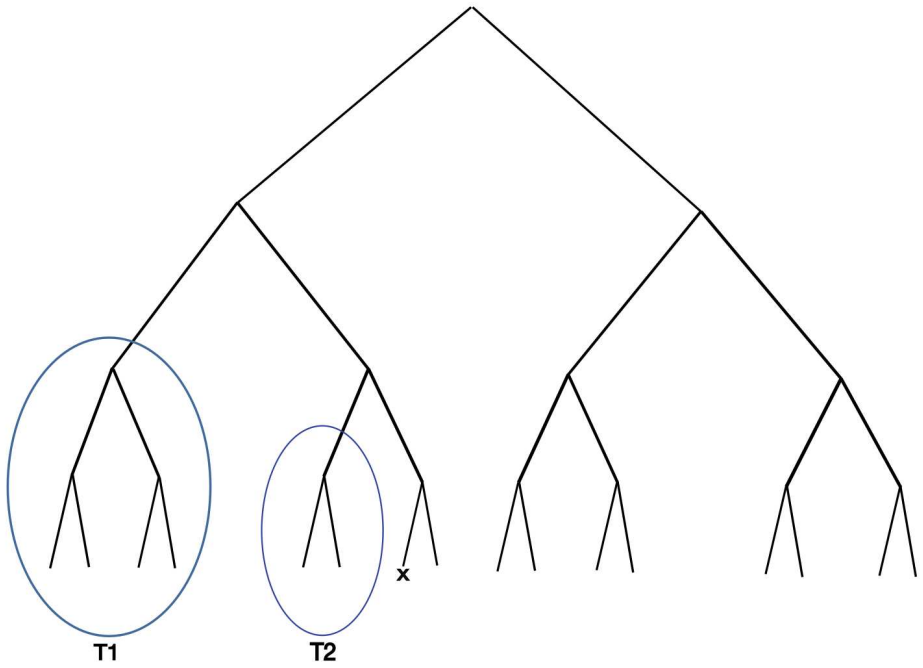

Figura 2.2: Decision Tree per la codifica aritmetica. In figura sono indicati i sotto-alberi la cui probabilità viene sommata per generare la codifica della sequenza x.

di una stringa e di conseguenza $F(x)$, che risulta

$$F(x^n) = \sum_{y^n \leq x^n} p(y^n) \tag{2.25}$$

La codifica avviene quindi *sequenzialmente*, così come la decodifica. Per questa parte, partiremo dalla probabilità di una certa sequenza per ricavarne il valore.

Partendo dal nodo radice, se la probabilità della sequenza è maggiore di $p(0)$, la sequenza inizierà con '1', e scorreremo il sotto-albero destro, sempre entrando nel destro per probabilità maggiore del nodo radice, e nel sinistro quando la probabilità è minore, per risalire alla sequenza completa.

Vediamo ora un esempio semplice per spiegare i concetti appena espressi. Il procedimento è illustrato graficamente in Figura 2.3. Ad ogni passo corrisponde una riga dell'immagine.

Prendiamo una sequenza di n variabili aleatorie indipendenti ed identicamente distribuite, ognuna con alfabeto $\mathcal{X} = \{0, 1\}$, $p(0) = 0.7$ e $p(1) = 0.3$. Vogliamo cercare la codifica della sequenza (01110).

Iniziando dalla prima riga della figura, cioè l'intervallo [0,1) (la funzione di distribuzione dalla quale ricaviamo la codifica assume solo valori tra 0 ed 1),

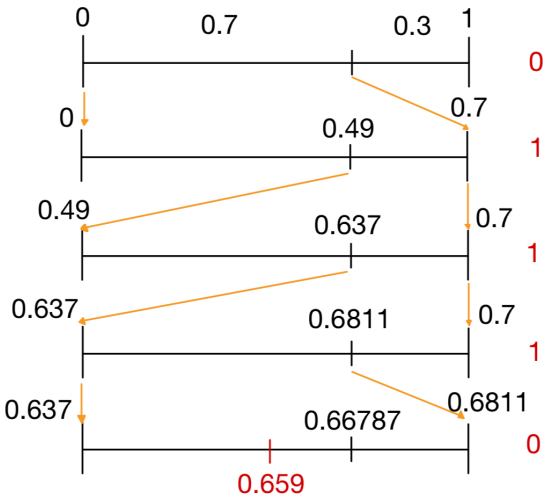

Figura 2.3: Illustrazione del processo di codifica aritmetica attraverso intervalli.

possiamo scendere ad ogni iterazione nella sotto-sequenza (sotto-albero) corrispondente al bit che vogliamo codificare: in questo modo andremo a considerare intervalli sempre più piccoli, che rappresentano la sequenza considerata fino a quella iterazione. In questo caso il primo bit da codificare è '0', il che significa che scenderemo nel sotto-albero sinistro. Alla seconda iterazione (seconda riga dell'immagine), l'intervallo viene diminuito a [0,0.7): questo sta a significare che il valore definito dalla funzione di distribuzione assegnato alla sequenza completa sarà sicuramente compreso in questo intervallo. Dato che dobbiamo codificare un '1', scendiamo nell'intervallo [0.49, 0.7) (sotto-albero destro) e ripetiamo il procedimento.

All'ultimo passaggio, avremo un intervallo che contiene tutta l'informazione necessaria per risalire alla sequenza. Possiamo ora calcolare il valore medio dell'intervallo "finale" ed utilizzarlo come codifica per la sequenza, allo stesso modo in cui abbiamo fatto per Shannon-Fano-Elias. In questo caso otteniamo $\frac{0.637+0.6811}{2} = 0.659$

Questo metodo, seppur più intuitivo, è equivalente al calcolare il valore della funzione di distribuzione di x^n:$F(01110) = p(00) + p(010) + p(0110)$.

Questa codifica è molto adatta per modelli le cui variabili sono indipendenti ed identicamente distribuite, cioè:

$$p(x^n) = \prod_{i=1}^{n} p(x_i) \tag{2.26}$$

La conoscenza completa della distribuzione è infatti fondamentale tanto per il

codificatore quanto per il decodificatore.

Capitolo 3

Asymptotic Equipartition Property

L'idea più importante che abbiamo ricavato nella prima parte del documento è questa: maggiore è l'entropia di un messaggio (ad esempio, un testo da comprimere), maggiore è l'informazione che esso ci porta, e maggiore è il numero di bit minimi richiesti per la sua compressione. Dato questo limite inferiore, che non può essere oltrepassato da nessun algoritmo di compressione *lossless*, introduciamo una proprietà che, oltre a confermare ciò che abbiamo già dimostrato, ci fornisce indicazioni sul comportamento di una sequenza di variabili aleatorie, mostrando come esso diventa sempre più prevedibile man mano che aumenta il numero di campioni che le appartengono.

3.1 Legge dei grandi numeri

Iniziamo richiamando un concetto fondamentale della teoria della probabilità. La legge dei grandi numeri (debole) descrive il comportamento in media di una sequenza di variabili aleatorie i.i.d. per un elevato numero di campioni. Data una successione di variabili aleatorie $X_1, X_2, \ldots, X_N$, consideriamo la media campionaria:

$$\overline{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{3.1}$$

Definizione 4. ***Legge debole dei grandi numeri:*** *Data una sequenza* $\{X_n\}$ *di variabili aleatorie i.i.d., ognuna con media statistica* $E(X_i) = \mu$*, allora, per ogni* $\epsilon > 0$*:*

$$\lim_{n \to \infty} P\left(\left|\overline{X}_n - \mu\right| < \epsilon\right) = 1 \tag{3.2}$$

A parole, questo teorema implica che la media campionaria, per n molto grande, tende in probabilità alla media statistica di una singola variabile aleatoria.

3.2 Asymptotic Equipartition Property

Formalmente, essa è definita come

Definizione 5. *Siano $X_1, X_2, \ldots, X_n$ variabili aleatorie indipendenti ed identicamente distribuite con distribuzione p(x), allora:*

$$-\frac{1}{n} logp(X_1, X_2, \ldots, X_n) \to H(X) \tag{3.3}$$

in probabilità.

Dimostrazione: *Poiché le variabili $X_1, X_2, \ldots, X_n$ sono indipendenti, possiamo scrivere:*

$$-\frac{1}{n} logp(X_1, X_2, \ldots, X_n) = -\frac{1}{n} \sum_{i=1}^{n} \log p(X_k) \tag{3.4}$$

Ora, per la definizione 4, sappiamo che la parte a destra dell'equazione, essendo una media campionaria, al crescere di n converge a

$$E(\log p(X_k)) = H(X) \tag{3.5}$$

e questo conclude la dimostrazione.

Ricordiamo che la convergenza in probabilità garantisce che, per n tendente all'infinito, cioè quando il numero di *campioni* presenti all'interno del set considerato è molto grande, la probabilità che la quantità a sinistra della freccia differisca dalla quantità a destra della freccia per più di una quantità arbitrariamente piccola ϵ è tendente ad 1, ovvero, in questo caso

$$\lim_{n\to\infty} P\left(| -\frac{1}{n} logp(X_1, X_2, \ldots, X_n) - H(X)| \right) = 1 \tag{3.6}$$

Da sola, questa proprietà non fornisce molte informazioni: essa è solamente una diretta applicazione della legge debole dei grandi numeri, che indica come la parte a sinistra converge, in probabilità, all'entropia della variabile X. Essa però risulta fondamentale per le prossime dimostrazioni.

3.3 Tipicità

Prima di procedere con l'analisi di ciò che la AEP comporta, è importante introdurre il concetto di **tipicità**. Iniziamo con un esempio: se esaminiamo una

stringa di bit di lunghezza N, in cui ogni elemento è la realizzazione di una variabile aleatoria, ognuna con la stessa probabilità (variabili aleatorie di Bernoulli di parametro p, indipendenti ed identicamente distribuite), otteniamo che la distribuzione di probabilità di X^N è una binomiale di parametri, appunto, N e p. In particolare, se consideriamo una certa realizzazione del messaggio X^N, in cui il numero di 1 sia r, otteniamo che la probabilità di questa particolare stringa è:

$$P(r) = \binom{N}{r} p^r (1-p)^{N-r} \tag{3.7}$$

Ciò che ci interessa maggiormente sono due caratteristiche del primo ordine di questa variabile aleatoria: media ($\mu = Np$) e deviazione standard ($\sigma = \sqrt{Np(1-p)}$). Notiamo che, al crescere di N, mentre la prima cresce linearmente con N stesso, la seconda cresce solamente con $\sqrt{N}$: questo comporta, al crescere di N, una "convergenza" sempre più marcata verso il valore medio. Come vedremo meglio più avanti, ciò che ci interessa è appunto il fatto che, data la convergenza, il messaggio ha una sempre maggior probabilità di appartenere ad un ristretto insieme di messaggi, che definiamo come **insieme tipico**.

Questo fatto sarà fondamentale per dimostrare alcune proprietà delle sequenze di simboli indipendenti ed identicamente distribuiti.

3.4 Insieme tipico e proprietà della AEP

Applicheremo ora il concetto di tipicità ad una più generale stringa di variabili aleatorie i.i.d., ognuna con alfabeto $\mathcal{X}$.

Una lunga sequenza di N variabili aleatorie X conterrà, all'incirca, $p_1 N$ occorrenze del primo simbolo (dove p_1 ne è la probabilità), $p_2 N$ occorrenze del secondo simbolo, e così via per tutti i simboli. Per questo motivo, la probabilità di una stringa tipica, è

$$P(\mathbf{x}) = P(x_1)P(x_2)\dots P(x_n) = p_1^{p_1 N} p_2^{p_2 N} \dots p_n^{p_n N} \tag{3.8}$$

Calcolando l'informazione di X^N otteniamo:

$$\log_2 \frac{1}{P(X)} = N \sum_i p_i \log_2 \frac{1}{p_i} = NH \tag{3.9}$$

NH è quindi l'informazione contenuta da una certa stringa $\mathbf{x}$ appartenente al set tipico. Dall'equazione 3.9 ricaviamo quindi che la probabilità di una sequenza

tipica è, all'incirca, 2^{-NH}.

Nota: Va specificato che tra le possibili realizzazioni della stringa di variabili aleatorie, alcune sequenze hanno probabilità maggiori. Si pensi ad esempio ad una variabile aleatoria di Bernoulli con probabilità $p(0) = 0.8$. Chiaramente, in questo caso, qualsiasi sia la lunghezza della stringa, la sequenza di soli zeri è quella con probabilità maggiore. Queste sequenze però non rientrano necessariamente nell'insieme tipico perché, come sarà mostrato più avanti, al crescere di N esse contribuiscono sempre meno alla probabilità totale.

Possiamo ora dare una definizione più formale di insieme tipico.

Definizione 6. *L'insieme tipico $A_\epsilon^{(n)}$ rispetto a p(x) è l'insieme di sequenze tale per cui:*

$$2^{-n(H(X)+\epsilon)} \leq p(x_1, x_2, \ldots, x_n) \leq 2^{-n(H(X)-\epsilon)} \tag{3.10}$$

In questo caso, ϵ è un parametro che può essere reso più piccolo all'aumentare di N, e che specifica quanto sia valida l'approssimazione della probabilità di un elemento dell'insieme tipico.

Dalla definizione (6), e ricordando la definizione (5), ricaviamo le seguenti 4 proprietà:

- Se $(x_1, x_2, \ldots, x_n) \in A_\epsilon^{(n)}$, allora $H(X) - \epsilon \leq -\frac{1}{n} logp(X_1, X_2, \ldots, X_n) \leq H(X) + \epsilon$
- $P(A_\epsilon^{(n)}) > 1 - \epsilon$
- $|A_\epsilon^{(n)}| \leq 2^{n(H(X)+\epsilon}$, con $|A|$ la cardinalità del set A
- $|A_\epsilon^{(n)}| \geq (1-\epsilon)2^{n(H(X)n-\epsilon)}$ per n sufficientemente grande

Dimostrazione: La prima proprietà deriva direttamente dalla definizione (3.10), utilizzando il logaritmo in base 2 e riarrangiando i termini.

Per la seconda, consideriamo $\lambda > 0$, allora, per la convergenza in probabilità della AEP, possiamo dire che esiste un n_0 tale che, per tutti gli $n \geq n_0$, vale:

$$P\left(\left|-\frac{1}{n} logp(X_1, X_2, \ldots, X_n - H(X)\right| < \epsilon\right) > 1 - \lambda. \tag{3.11}$$

Ponendo ora $\lambda = \epsilon$ otteniamo la dimostrazione. Dato che appunto abbiamo già visto come sia l'insieme tipico a soddisfare l'equazione della AEP, concludiamo che

la probabilità a sinistra della diseguaglianza sia proprio la probabilità dell'intero insieme.

Per ottenere la terza, scriviamo

$$1 = \sum_{\mathbf{x} \in X^n} p(\mathbf{x}) \tag{3.12}$$

$$\geq \sum_{\mathbf{x} \in A_\epsilon^{(n)}} p(\mathbf{x}) \tag{3.13}$$

$$\geq \sum_{\mathbf{x} \in A_\epsilon^{(n)}} 2^{-n(H(X)+\epsilon)} \tag{3.14}$$

$$2^{-n(H(X)+\epsilon)} |A_\epsilon^{(n)}| \tag{3.15}$$

Questo calcolo, in particolare il secondo passaggio, è ottenuto dalla definizione di insieme tipico, mentre l'ultimo passaggio conclude la dimostrazione.

Per l'ultima, consideriamo

$$1 - \epsilon < P(A_\epsilon^{(n)}) \tag{3.16}$$

$$\leq \sum_{\mathbf{x} \in A_\epsilon^{(n)}} 2^{-n(H(X)-\epsilon)} \tag{3.17}$$

$$2^{-n(H(X)+\epsilon)} |A_\epsilon^{(n)}| \tag{3.18}$$

Da cui si ricava immediatamente la formula da dimostrare.

3.5 Conseguenza sulla compressione dati

Queste proprietà ci danno un risultato veramente importante. Esse dicono che quando il campione di variabili aleatorie, o, considerando un messaggio, *il numero di simboli che lo compongono*, è molto grande, le sequenze che appartengono all'insieme tipico, cioè quelle che appaiono con probabilità indicata nella equazione

(3.10), sono equiprobabili e, soprattutto, la probabilità che il messaggio considerato sia effettivamente presente in questo insieme è tendente ad 1. Come si era accennato prima, infatti, nel calcolo della probabilità totale, alcune sequenze molto probabili, essendo presenti in quantità molto esigua, non influiscono nel fatto che la probabilità dell'insieme tipico tenda ad 1.

La conseguenza per la compressione dei dati è immediata. Come si può ricordare dalla precedente trattazione riguardante la codifica entropica, l'obiettivo degli algoritmi di codifica consiste nell'assegnare *i simboli più probabili alle sequenze più brevi della codifica*, in modo da garantirne efficienza diminuendo il numero medio di bit richiesti. La AEP garantisce quindi l'esistenza di un insieme di sequenze (l'insieme tipico) che sappiamo con certezza avranno probabilità molto grande di apparire, se considerate come insieme.

3.5.1 Relazione con il teorema di Shannon

Consideriamo gli insiemi di sequenze $X_1, X_2, \ldots, X_n$ appartenenti allo spazio X^n. Lo dividiamo in due set: il set tipico ed il suo complementare.

Se ordiniamo le sequenze, come ad esempio fatto per la codifica aritmetica, possiamo rappresentarle attraverso degli indici.

Dato che abbiamo $\leq 2^{n(H+\epsilon)}$ sequenze nel set tipico, abbiamo bisogno di non più di $n(H+\epsilon)+1$ bit per l'indicizzazione (il $+1$ nel caso $n(H+\epsilon)$ non sia intero). Aggiungendo il bit '0' all'inizio di ogni sequenza del set tipico, ed '1' a tutte le altre sequenze, che quindi richiedono sicuramente meno di $nlog|X| + 2$ bit per essere rappresentate, otteniamo un codice per **tutte** le sequenze possibili.

Nota: $nlog|X| + 2$ sono i bit necessari a rappresentare tutte le sequenze possibili, ma dato che il complementare del set tipico è un sottoinsieme delle sequenze possibili, useremo sicuramente meno bit per rappresentarlo.

Il risultato è che abbiamo ottenuto un codice per cui valgono le seguenti proprietà:

- Esso è One-to-one, e facilmente decodificabile, poiché il primo bit di ogni simbolo ne indica la lunghezza

- le sequenze **più comuni** hanno lunghezza $\approx nH$

Calcoliamo ora il valore atteso della lunghezza del codice ottenuto:

$$
\begin{aligned}
E(l(X^n)) &= \sum_{x \in X^n} p(x^n) l(x^n) \\
&= \sum_{x^n \in A_\epsilon^{(n)}} p(x^n)\, l(x^n) + \sum_{x^n \in (A_\epsilon^{(n)})^c} p(x^n) l(x^n) \\
&\leq \sum_{x^n \in A_\epsilon^{(n)}} p(x^n)[n(H+\epsilon)+2] + \sum_{x^n \in (A_\epsilon^{(n)})^c} p(x^n)(n \log |\mathcal{X}| + 2) \\
&= \Pr\{A_\epsilon^{(n)}\}[n(H+\epsilon)+2] + \Pr\{(A_\epsilon^{(n)})^c\}(n \log |\mathcal{X}| + 2) \\
&\leq n(H+\epsilon) + \epsilon n(\log |\mathcal{X}|) + 2 \\
&= n(H+\epsilon').
\end{aligned}
$$

Il parametro $\epsilon' = \epsilon = \log |\mathcal{X}| + \frac{2}{n}$ può essere reso arbitrariamente piccolo tramite appropriata scelta di ϵ ed n. Perciò possiamo dire che in media utilizzeremo $nH(X)$ bit per rappresentare le sequenze.

Questo è proprio il cosiddetto "Teorema di codifica di sorgente di Shannon". Esso afferma che date N variabili aleatorie X, i.i.d., con entropia $H(X)$, esse possono essere compresse utilizzando $NH(X)$ bit, con perdita di informazione trascurabile, per $N \to \infty$; mentre, se esse sono convertite in un messaggio con meno di $NH(X)$ bit, si avrà perdita di informazione.

In modo più formale, possiamo scrivere

Teorema 4. *Data una sequenza di variabili aleatorie i.i.d. X^n ed $\epsilon > 0$, esiste un codice che mappa le sequenze x^n in stringhe binarie tale che*

$$
E\left[\frac{1}{n} l(X^n)\right] \leq H(X) + \epsilon \tag{3.19}
$$

quando n è sufficientemente grande.

Cioè possiamo rappresentare sequenze X^n usando *in media* NH bit.

Capitolo 4

Esempio di implementazione

Dopo aver analizzato a livello teorico come ci aspettiamo che si comportino gli algoritmi di compressione, in questo ultimo capitolo proveremo a fornire un semplice esempio del loro funzionamento.

Analizzeremo due algoritmi: la codifica di Huffman e una codifica basata esclusivamente sui principi della AEP.

4.1 Codici

Di seguito viene fornita l'implementazione in Python degli algoritmi. Entrambi i codici sono riportati nelle rispettive appendici. Nei paragrafi seguenti sono riportati dei commenti riguardo al loro funzionamento. Ogni algoritmo viene testato con una sequenza di bit generata casualmente, e con lunghezze di simbolo diverse, ma con distribuzione di probabilità sempre costante: $p(0) = 0.8$, $p(1) = 0.1$.

4.1.1 Codice di Huffman

Per questa implementazione sono state utilizzate solamente strutture dati basilari: dizionari, liste e alberi binari. L'unica libreria importata è "random", per la generazione casuale della stringa.

Il codice inizia generando la stringa di 'zeri' ed 'uni'. Essa è quindi suddivisa in simboli, che vengono aggiunti, insieme alle loro frequenze (cioè le loro probabilità), come coppia chiave - valore, ad un dizionario.

Successivamente viene seguito strettamente l'algoritmo specificato dalla codifica di Huffman: il dizionario viene man mano svuotato per creare un albero

binario. Questo albero è quindi utilizzato per generare una codifica, dove ogni simbolo del messaggio originale è associato ad un simbolo dell'alfabeto di codifica. Infine viene creata la stringa codificata.

Per la decodifica, invece, il passaggio è più semplice e prevede di risalire al messaggio originale guardando alla stringa codificata e conoscendo il dizionario di codifica (associazione simbolo originale - simbolo codificato).

Essendo il codice *prefix-free*, è solamente necessario iterare lungo la stringa codificata e, ogni volta che viene riconosciuto un simbolo, associarlo immediatamente al simbolo originale, costruendo ad ogni passo la stringa sorgente. Infine vengono stampati tutti i dati relativi al programma: stringa originale, dizionario di codifica, messaggio codificato, rapporto di compressione e messaggio decodificato (che deve essere identico al messaggio originale).

La complessità dell'algoritmo non è studiata ma può sicuramente essere migliorata utilizzando strutture dati ed algoritmi più efficienti, ma per gli esempi proposti il programma rimane comunque molto veloce.

4.1.2 Codifica tramite AEP

Come per il precedente algoritmo, non sono state utilizzate strutture dati se non quelle implementate di default in Python (in questo caso, solo liste e dizionari). Le uniche librerie importate sono utilizzate per alcuni calcoli matematici richiesti dalla codifica.

Iniziamo generando casualmente la sequenza di bit, e la suddividiamo in simboli di lunghezza fissa.

Dopodiché seguiamo strettamente i passaggi dati dalla teoria: calcoliamo le probabilità di ogni simbolo (sequenza di zeri ed uni), in modo da creare la divisione in sequenze tipiche e sequenze atipiche.

Per questi due insiemi calcoliamo il numero di bit necessari a codificarli ("parte alta" del logaritmo in base due del numero di sequenze).

Ora, per ogni simbolo appartenente alla stringa da codificare, ne calcoliamo una rappresentazione binaria in questo modo: se il simbolo appartiene all'insieme tipico, convertiamo in binario l'indice che gli corrisponde (se è il primo simbolo che abbiamo trovato, l'indice è '0' convertito in binario, se è il secondo simbolo, abbiamo '1' convertito in binario, e così via), assicurandoci che la lunghezza della stringa sia uguale per ogni simbolo (cioè, come detto prima, la "parte alta" del logaritmo del numero di sequenze), aggiungendo zeri a sinistra delle sequenze più corte. Infine aggiungiamo uno zero o un uno (a seconda che la sequenza

appartenga all'insieme tipico o atipico) a sinistra di ogni sequenza, che servirà "bandierina" per dirci che il simbolo codificato corrisponde ad una sequenza tipica o atipica.

Per la decodifica, come per il precedente algoritmo, sarà sufficiente scorrere la stringa codificata e far corrispondere ogni simbolo trovato alla sequenza originale, tramite una mappa di decodifica.

4.1.3 Considerazioni

Lo scopo dei due algoritmi è diverso: se il primo, cioè la codifica di Huffman, mostra un semplice esempio di come possa essere veramente realizzata una codifica in modo "pratico", cioè attraverso un noto ed efficienze algoritmo, il secondo ha il solo scopo di provare la teoria dell'*Asymptotic Equipartition Property*, e della tipicalità.

Esso infatti, sebbene possa essere utilizzato come algoritmo di codifica, non è paragonabile al primo né in termini di complessità computazionale, né in termini di rapporto di compressione (e quindi di efficienza).

La sua analisi si concentra infatti sul dimostrare come un insieme molto ridotto (rispetto al totale) di sequenze, possa racchiudere al suo interno la maggior parte della massa di probabilità, e su come, da alcuni semplici principi, si possa effettivamente realizzare una molto rudimentale compressione dati.

4.2 Risultati

Analizziamo ora i risultati ottenuti dall'analisi dei due algoritmi.

4.2.1 Codice di Huffman

Nell'esperimento considerato, la distribuzione di probabilità rimane fissata. La lunghezza del messaggio è fissata a 1000000 di bit, per garantire che la maggior parte delle sequenze possibili venga effettivamente generata dal programma e che la distribuzione delle stesse sia quanto più possibile vicina a quella teorica. Ogni punto è associato ad una lunghezza di simboli diversa. Per ottenere una maggiore precisione, con ogni lunghezza vengono effettuate 10 iterazioni dell'algoritmo e l'output è una media del valore atteso della lunghezza della codifica.

Nota: la lunghezza totale non è realmente fissata. Per garantire che la codifica funzioni correttamente, essa deve sempre essere un multiplo della lunghezza

di simbolo, in modo che quando andiamo a suddividerla, ogni simbolo abbia la stessa lunghezza. Se ad esempio avessimo lunghezza totale uguale a 1000000 e lunghezza del simbolo uguale a 3, dividendo il messaggio, otterremmo sempre un unico simbolo di lunghezza 1 (lunghezza data dal resto della divisione tra 1000000 e 3). Questo andrebbe a modificare leggermente la codifica, creando un ulteriore simbolo. Per questo motivo, la lunghezza viene scelta come multiplo della lunghezza di simbolo, ma rimane sempre attorno ai 1000000 bit. Ad esempio per lunghezza simbolo uguale a 3, la lunghezza del messaggio totale è in realtà 999999 bit. Questa idea viene mantenuta anche nella prossima codifica.

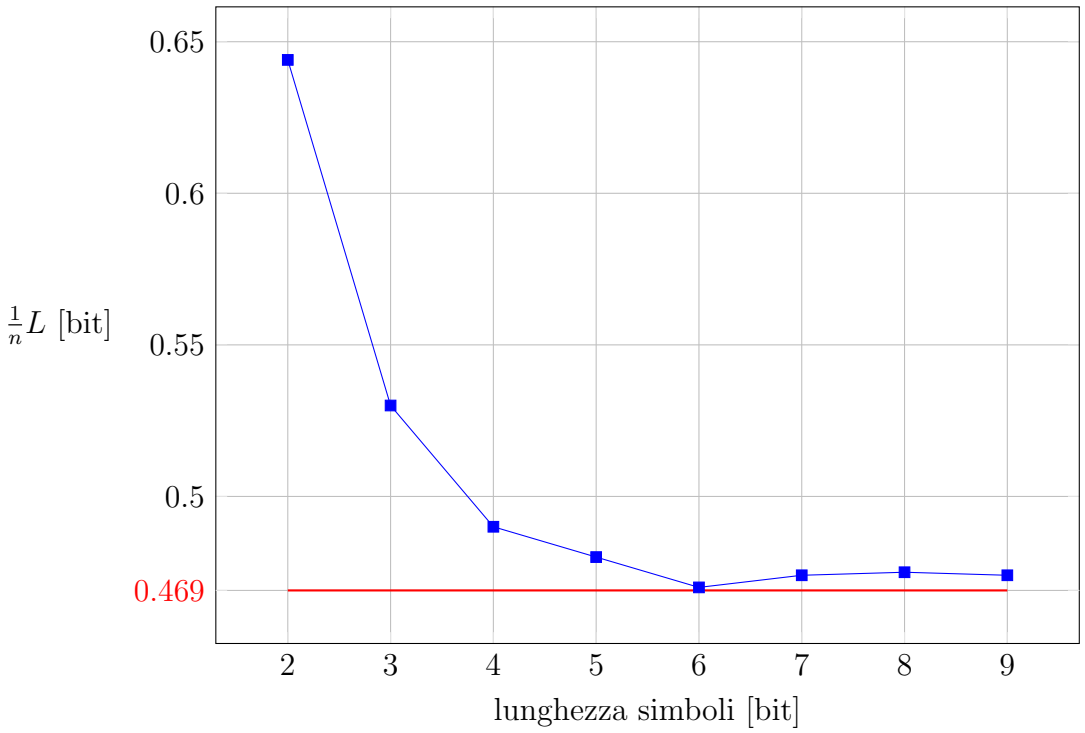

Figura 4.1: Grafico della lunghezza media dei simboli nella codifica tramite codifica di Huffman.

La Figura 4.1 mostra il valore atteso della lunghezza del codice, diviso per la lunghezza di simbolo n in modo da poter fare un confronto con l'entropia di sorgente, il cui valore è evidenziato da una linea rossa. In accordo con il Teorema di Shannon per la codifica di sorgente a blocchi, il valore atteso dei simboli di codice tende sempre di più verso l'entropia della sorgente, senza però oltrepassarla. Ovviamente questo non viene senza un costo: più lunghi sono i simboli e maggiore è la loro quantità, con un aumento esponenziale. Ad esempio, con lunghezza dei simboli 3, avremo $2^3 = 8$ simboli disponibili, con lunghezza 9, i

simboli possibili sono invece $2^9 = 512$. Questo porta ad una maggiore complessità della mappa di codifica, poiché molti più simboli possono essere generati e di conseguenza associati ad un simbolo di codifica.

4.2.2 Codifica tramite AEP

Nei seguenti grafici, invece dell'efficacia della compressione, andiamo a verificare l'idea della tipicità. Quello su cui ci concentriamo è la prima parte del programma: generazione delle sequenze e suddivisione in insieme tipico ed insieme atipico.

Per tutti i test il parametro ϵ è fissato a $0,2$. Ciò che ci interessa verificare è la proprietà per cui, all'aumentare della lunghezza dei simboli, l'insieme tipico conterrà, pur mantenendo una cardinalità relativamente piccola, la maggior parte della densità di probabilità.

Nello specifico, la Figura 4.2 mostra le densità di probabilità contenute in entrambi gli insiemi. In blu è mostrata la probabilità contenuta nell'insieme tipico, in rosso quella dell'insieme atipico. Nelle Figure da 4.3 a 4.6 invece, vengono mostrate le cardinalità degli insiemi.

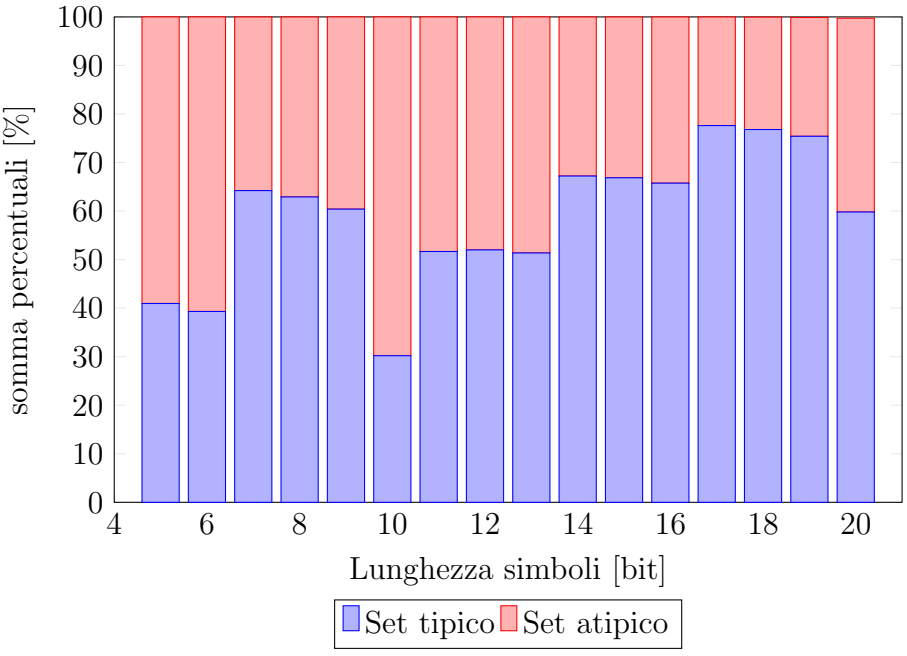

Figura 4.2: Grafico distribuzione set tipico e atipico (1000000000 bit).

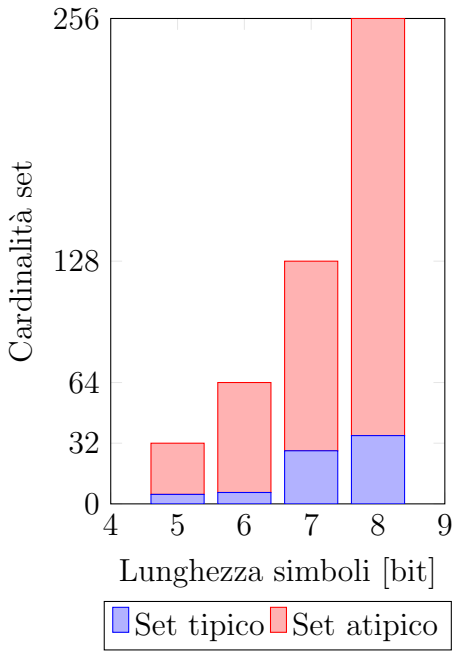

Figura 4.3: Cardinalità (5-8)

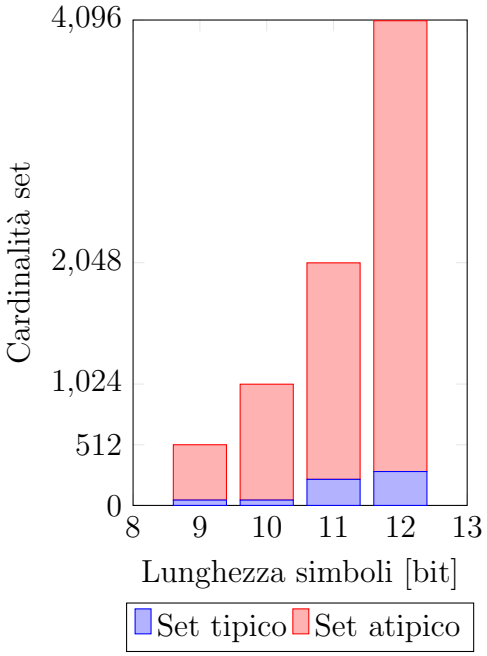

Figura 4.4: Cardinalità (9-12)

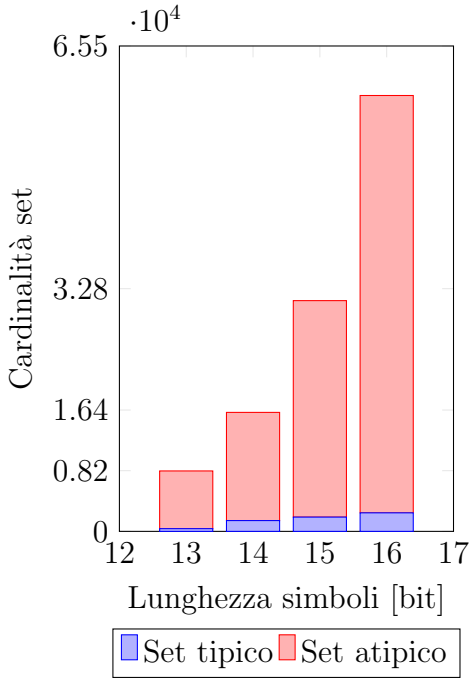

Figura 4.5: Cardinalità (13-16)

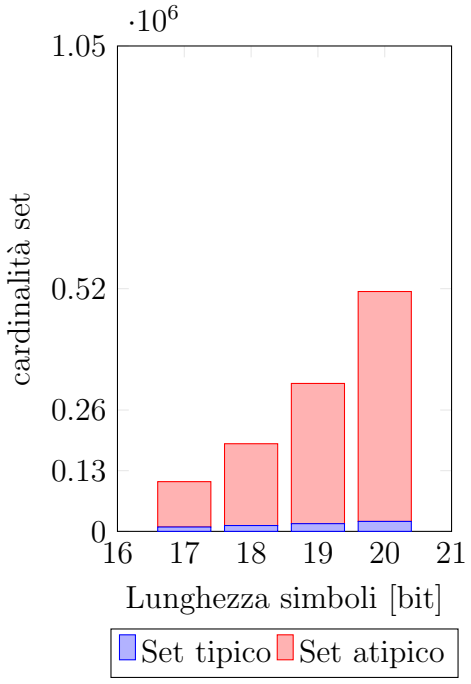

Figura 4.6: Cardinalità (17-20)

Innanzitutto notiamo che, anche se le due cardinalità dovrebbero tutte sommare a 2^n, dove n è la lunghezza del simbolo, per le ultime sequenze (quelle più lunghe, appartenenti all'ultimo grafico), la somma è decisamente minore rispetto a quello che ci aspettiamo. Il motivo di ciò è che, data la lunghezza finita della sequenza generata (molto grande, 1000000000 bit, ma comunque finita), il programma non genera tutte le sequenze possibili: quelle meno probabili non vengono mai generate. Questo fatto però non cambia l'analisi: come vediamo dal primo grafico, anche le sequenze più lunghe hanno probabilità totale che somma a 100%, perché le sequenze non generate contengono una probabilità infinitesima. Possiamo quindi considerare questi numeri come una buona approssimazione.

Ciò che ricaviamo dai grafici è in pieno accordo con la teoria. Le sequenze tipiche, pur rimanendo un numero molto piccolo rispetto al totale delle sequenze generate, contengono, all'aumentare della lunghezza del simbolo, una quantità di massa di probabilità sempre maggiore. Negli esempi proposti, questa probabilità arriva quasi a toccare l'80% per le sequenze più lunghe.

Questo implica, come avevamo già dimostrato a livello teorico nel capitolo riguardante la *Asymptotic Equipartition Property*, che durante la codifica potremo assegnare simboli corti a sequenze che hanno probabilità estremamente alta di essere generate diminuendo, come è obiettivo di un buon algoritmo di compressione, la lunghezza media della codifica.

Per concludere possiamo effettuare per questa codifica la stessa analisi già fatta per la codifica di Huffman. In Figura 4.7 mostriamo un grafico nel quale vengono rappresentate l'entropia della singola variabile aleatoria X e la lunghezza media della codifica. Questo grafico viene realizzato con lunghezza totale del messaggio 10000000 bit. Sebbene i risultati per questa codifica siano peggiori rispetto a quelli della codifica di Huffman, anche qui notiamo la tendenza della curva verso la linea rossa corrispondente all'entropia. Questo conferma nuovamente i risultati teorici discussi precedentemente.

4.2.3 Conclusioni

I risultati ottenuti verificano chiaramente la validità teorica della AEP e del teorema di codifica a blocchi di Shannon. Per la prima abbiamo evidenziato come si distribuiscono le sequenze generate tra i due insiemi tipico ed atipico verificando che questo avviene in accordo con la teoria. Per il secondo invece la verifica è stata fatta con due algoritmi di compressione diversi mostrando come entrambi portano a risultati simili.

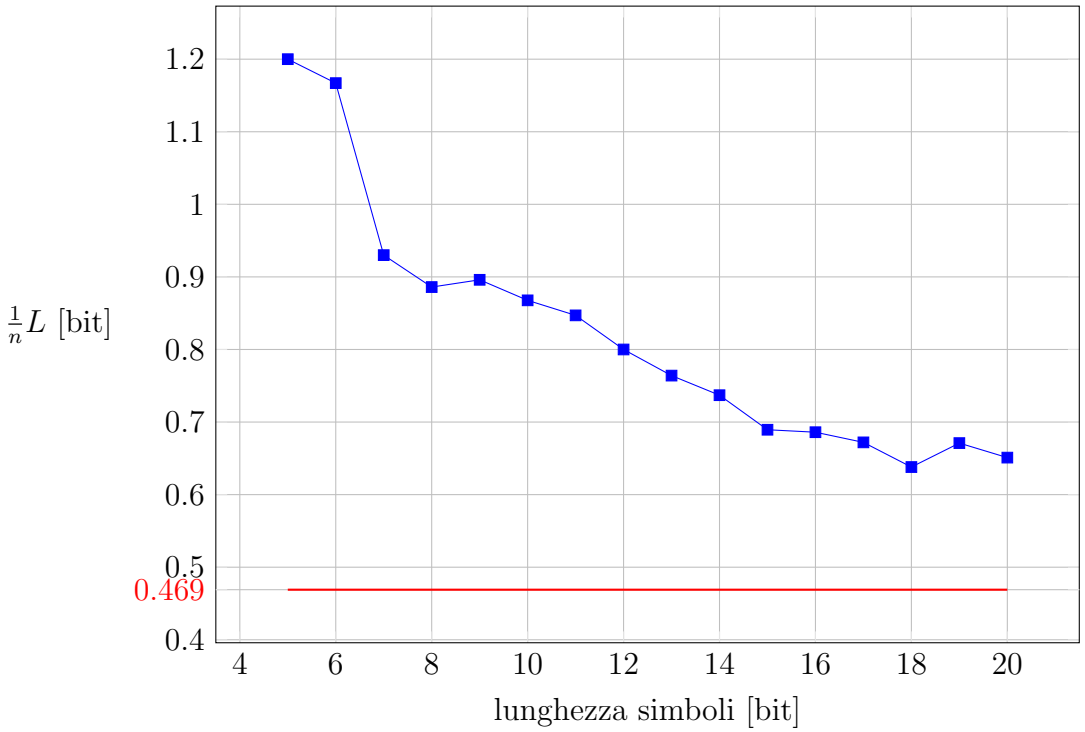

Figura 4.7: Grafico della lunghezza media dei simboli nella codifica tramite *Asymptotic Equipartition Property*.

Quello che ne possiamo ricavare è che sebbene i codici siano semplici, l'efficienza delle implementazioni lontana da quella di algoritmi realmente utilizzati nella compressione dati e la lunghezza dei simboli lontana dal valore "infinito" richiesto dalla teoria, si può notare chiaramente la tendenza di miglioramento delle prestazioni all'aumentare della lunghezza dei simboli.

Bibliografia

[1] Thomas M. Cover, Joy A. Thomas, 1991, New York, Wiley, *Elements of information theory*

[2] Khalid Sayood, 2006, University of Nebraska, *Introduction to Data Compression*

[3] David J.C. MacKay, *Information Theory, Inference, and Learning Algorithms*

[4] Raymond W. Yeung, 2002, *A First Course in Information Theory*

[5] Lorenzo Finesso, 2022, *Lezioni di Probabilità*

[6] Nevio Benvenuto, Michele Zorzi, *Principles of Communications Networks and Systems*

Appendice A: Codice Huffman

```
import random
from math import log

simbol_length = 5
data_length = 1000000
simbols = data_length / simbol_length
p0 = 0.9

class Nodo:
    def __init__(self, nodo, sinistro = None, destro = None):
        self.nodo = nodo
        self.sinistro = sinistro
        self.destro = destro

    def print(self):
        if self.sinistro is not None:
            self.sinistro.print()
        print(self.nodo)
        if self.destro is not None:
            self.destro.print()

    def get_sinistro(self):
        return self.sinistro

    def get_destro(self):
        return self.destro

    def get_nodo(self):
        return self.nodo

    def isLeaf(self):
        return self.sinistro is None and self.destro is None

def entropy(p0):
    return p0*(log(1/p0, 2)) + (1 - p0)*log(1/(1 - p0), 2)
def random_generator(data_length, p0):
    p1 = 1 - p0
    bits = ['0', '1']
    probabilities = [p0, p1]
    bitstring = ''.join(random.choices(bits, probabilities, k=
     data_length))
    return bitstring
```

```
def get_min_val(dict):
    freq_min = list(dict.values())[0]
    freq_min2 = list(dict.values())[1]
    key_min = list(dict.keys())[0]
    key_min2 = list(dict.keys())[1]
    for i, element in enumerate(dict):
        if i >= 2:
            max_freq_min = max(freq_min, freq_min2)
            freq = dict[element]
            if freq < freq_min or freq < freq_min2:
                if max_freq_min == freq_min:
                    freq_min = freq
                    key_min = element
                elif max_freq_min == freq_min2:
                    freq_min2 = freq
                    key_min2 = element
    del dict[key_min]
    del dict[key_min2]
    dict[key_min + key_min2] = freq_min + freq_min2
    return (key_min, freq_min), (key_min2, freq_min2), ((key_min
     + key_min2), (freq_min + freq_min2)), dict

def costruisci_albero(dict):
    supporto = []
    min1, min2, sum_min, dict = get_min_val(dict)
    Figlio_sinistro = Nodo(min1)
    Figlio_destro = Nodo(min2)
    Albero_temp = Nodo(sum_min, Figlio_sinistro, Figlio_destro)
    supporto.append(Albero_temp)
    while len(dict) > 1:
        min1, min2, sum_min, dict = get_min_val(dict)
        Figlio_sinistro = None
        Figlio_destro = None
        index1 = None
        index2 = None
        for i in range(len(supporto)):
            if supporto[i] is not None:
                if supporto[i].nodo == min1:
                    Figlio_sinistro = supporto[i]
                    index1 = i
                elif supporto[i].nodo == min2:
                    Figlio_destro = supporto[i]
                    index2 = i
```

```
        if Figlio_sinistro is None:
            Figlio_sinistro = Nodo(min1)
        if Figlio_destro is None:
            Figlio_destro = Nodo(min2)
        if index1 is not None and index2 is None:
            supporto.pop(index1)
        elif index2 is not None and index1 is None:
            supporto.pop(index2)
        elif index1 is not None and index2 is not None:
            indexes = [index1, index2]
            for index in sorted(indexes, reverse=True):
                supporto.pop(index)
        Albero_temp = Nodo(sum_min, Figlio_sinistro,
 Figlio_destro)
        supporto.append(Albero_temp)
    Albero = supporto[0]
    return Albero

def get_encoding(Albero, encoding, stringa = None):
    if stringa is None:
        stringa = ''
    if Albero.sinistro is not None:
        get_encoding(Albero.sinistro, encoding, stringa + '0')
    if Albero.isLeaf():
        encoding[list(Albero.get_nodo())[0]] = stringa
    if Albero.destro is not None:
        get_encoding(Albero.destro, encoding, stringa + '1')
    return

def huffman_coding(data):
    freqs = {}
    for i in range(0, data_length, simbol_length):
        simbol = data[i:i+simbol_length]
        if simbol not in freqs:
            freqs[simbol] = 1
        else:
            freqs[simbol] += 1
    Albero = costruisci_albero(freqs)
    encoding = {}
    get_encoding(Albero, encoding)
    messaggio_codificato = ''
    for i in range(0, data_length, simbol_length):
        simbol = data[i:i+simbol_length]
        codice = encoding[simbol]
```

```
            messaggio_codificato = messaggio_codificato + codice

    return messaggio_codificato, encoding

def huffman_decoding(messaggio_codificato, encoding):
    freqs = {}
    messaggio_decodificato = ''
    valore_atteso = 0
    i = 0
    j = i + 1
    temp = list(encoding.values())
    while i < len(messaggio_codificato):
        for element in temp:
            if element == messaggio_codificato[i:j]:
                chiave = next((k for k, v in encoding.items() if
 v == element), None)
                messaggio_decodificato = messaggio_decodificato +
  chiave
                element = int(element)
                if element not in freqs:
                    freqs[element] = 1
                else:
                    freqs[element] += 1
                i = j
                break
        j = j + 1
    for freq in freqs:
        freqs[freq] = freqs[freq] / simbols
    for element in temp:
        valore_atteso = valore_atteso + freqs[int(element)]*len(
 element)
    return messaggio_decodificato, freqs, valore_atteso
```

Appendice B: Codice AEP

```
import random
from math import log, ceil

p0 = 0.9
simbol_length = 19
data_length = 9999985
simbols = data_length / simbol_length
```

```
epsilon = 0.2

def random_generator(data_length, p0):
    p1 = 1 - p0
    bits = ['0', '1']
    probabilities = [p0, p1]
    bitstring = ''.join(random.choices(bits, probabilities, k=
     data_length))
    return bitstring

def simbol_generator(bitstring):
    global data_length, simbol_length
    freqs = {}
    for i in range(0, data_length, simbol_length):
        simbol = bitstring[i:i + simbol_length]
        if simbol not in freqs:
            freqs[simbol] = 1
        else:
            freqs[simbol] += 1
    return freqs

def entropy(p0):
    return p0*(log(1/p0, 2)) + (1 - p0)*log(1/(1 - p0), 2)

def get_prob(list):
    global p0
    probabilities = []
    for item in list:
        probabilities.append(pow( p0 , item.count('0') ) * pow(
     (1-p0) , item.count('1') ))
    return probabilities

def typical_sequences(dict):
    global data_length, simbol_length, epsilon
    H = entropy(p0)
    typ_seq = []
    atyp_seq = []
    typ_prob = []
    atyp_prob = []
    sequences = list(dict.keys())
    probabilities = get_prob(sequences)
    for (i, probability) in enumerate(probabilities):
        if probability >= pow(2, -(simbol_length*(H + epsilon)))
     and probability <= pow(2, -(simbol_length*(H - epsilon))):
```

```
            typ_seq.append(sequences[i])
            typ_prob.append(probability)
        else:
            atyp_seq.append(sequences[i])
            atyp_prob.append(probability)

    sum_typ = 0
    for item in typ_prob:
        sum_typ += item

    sum_atyp = 0
    for item in atyp_prob:
        sum_atyp += item

    return typ_seq, atyp_seq

def aep_coding(data):
    global p0
    freqs = simbol_generator(data)
    bit_typ = 0
    bit_atyp = 0
    typ_seq, atyp_seq = typical_sequences(freqs)
    if(len(typ_seq) != 0):
        bit_typ = ceil(log(len(typ_seq), 2))
    if(len(atyp_seq) != 0):
        bit_atyp = ceil(log(len(atyp_seq), 2))
    codifica = ''
    coding_map = {}
    i = 0
    j = 0
    for l in range(0, data_length, simbol_length):
        simbolo = data[l:l+simbol_length]
        if len(typ_seq) != 0 and simbolo in typ_seq:
            if simbolo not in coding_map:
                temp = '0' + format(i, f'0{bit_typ}b')
                codifica = codifica + temp
                coding_map[simbolo] = temp
                i += 1
            else:
                codifica = codifica + coding_map[simbolo]
        elif len(atyp_seq) != 0 and simbolo in atyp_seq:
            if simbolo not in coding_map:
                temp = '1' + format(j, f'0{bit_atyp}b')
                codifica = codifica + temp
```

```
                coding_map[simbolo] = temp
                j += 1
            else:
                codifica = codifica + coding_map[simbolo]
    return typ_seq, atyp_seq, coding_map, codifica, bit_typ,
     bit_atyp

def aep_decoding(coding_map, data, bit_typ, bit_atyp):
    freqs = {}
    i = 0
    decodifica = ''
    valore_atteso = 0
    temp = list(coding_map.values())
    while i < len(data):
        simbolo = None
        if data[i] == '0':
            simbolo = data[i : i + bit_typ + 1]
            chiave = next((k for k, v in coding_map.items() if v
     == simbolo), None)
            decodifica = decodifica + chiave
            i = i + bit_typ + 1
        elif data[i] == '1':
            simbolo = data[i : i + bit_atyp + 1]
            chiave = next((k for k, v in coding_map.items() if v
     == simbolo), None)
            decodifica = decodifica + chiave
            i = i + bit_atyp + 1
        if simbolo not in freqs:
            freqs[simbolo] = 1
        else:
            freqs[simbolo] += 1
    for freq in freqs:
        freqs[freq] = freqs[freq] / simbols
    for element in temp:
        valore_atteso = valore_atteso + freqs[element]*len(
     element)

    return decodifica, valore_atteso
```