

Multimedia

Homework 1

Alessandro Trigolo

30 Aprile 2024

Indice

1	Codifica semplice	4
1.1	Caricamento dell'immagine	4
1.2	Entropia dell'immagine	4
1.3	Codifica con dizionario	6
1.4	Discussione risultati parziali	6
1.5	Codifica semplice	7
1.6	Entropia dell'errore di predizione	8
1.7	Exp Golomb	8
1.8	Conclusioni	10
2	Codifica avanzata	11
2.1	Codifica avanzata	11

Elenco delle figure

1	Estrazione della luminanza da una immagine a colori.	5
2	Rappresentazione del modulo dell'errore di predizione nella codifica semplice.	8
3	Rappresentazione del modulo dell'errore di predizione nella codifica avanzata.	12
4	Rappresentazione della differenza tra l'errore della codifica semplice e della codifica avanzata.	13

Todo list

Inserisci introduzione dove spieghi l'obiettivo discuti brevemente come hai fatto	
il codice e dove trovarlo. Inserisci link a github etc	4
inserisci mini introduzione	4
capisci cosa significa il valore EG-bpp	8
forse ho detto una cazzata	10
Tutti i valori della codifica avanzata sono essenzialmente corretti in quanto sono	
minori rispetto alla codifica semplice	12

Introduzione

Il linguaggio scelto per completare le richieste dell'homework è **Python**; all'interno del documento saranno presenti solo i punti salienti dello script, che comunque può essere ispezionato al seguente [link](#).

Inserisci introduzione dove spieghi l'obiettivo discuti brevemente come hai fatto il codice e dove trovarlo. Inserisci link a github etc

inserisci mini introduzione

1 Codifica semplice

1.1 Caricamento dell'immagine

La prima richiesta dell'homework consiste nel caricare un'immagine a livelli di grigio oppure un'immagine a colori ed estrarne la luminanza, approssimabile come la media tra i tre canali di colori dell'immagine (*Red*, *Green* e *Blue*). Il frammento di codice seguente incontra esattamente le richieste della prima task dove, attraverso la funzione `imread` del modulo `matplotlib.image`, l'immagine viene letta correttamente ed, eventualmente, ne viene estratta la luminanza.

```
1  # Prepare to load the image
2  img_file_name = "spiderman"
3  img_extension = ".jpg"
4  current_dir = os.getcwd()
5
6  # path to reach the img
7  path_to_img = os.path.join(current_dir, "multimedia", "hw-1",
8                               "script", "imgs") + "/"
9
10 # loads the colored image
11 gray_img = mpimg.imread(path_to_img + img_file_name +
12                           img_extension).astype(np.int16)
13
14 # extracts the luminance if RGB
15 if gray_img[0][0].size > 1:
16     gray_img = np.dot(gray_img[..., :3], [1, 1, 1]) / 3
```

Scegliendo un'immagine a colori è quindi possibile verificare il corretto funzionamento del codice, come mostrato nella figura 1 dove si è scelto come riferimento quella di *Spiderman*.

1.2 Entropia dell'immagine

La seconda task dell'homework richiede di calcolare l'entropia dell'immagine in scala di grigi. L'entropia di una variabile aleatoria X (in questo caso l'immagine) è definita come l'**informazione media** degli eventi della sorgente; l'informazione di un evento è descritta dalla funzione seguente:

$$I(X) = \log_2 \left(\frac{1}{p_i} \right)$$



Figura 1: Estrazione della luminanza da una immagine a colori.

L'informazione media diminuisce all'aumentare della probabilità dell'evento p_i . Questo è ragionevole in quanto più un evento è improbabile (quindi $p_i \rightarrow 0$) e più la sua informazione è alta ($I(X) \rightarrow +\infty$). Assumendo che gli eventi della sorgente siano indipendenti, l'informazione media si traduce nella seguente formula:

$$H(X) = E[I(X)] = \sum_{i=1}^M p_i \log_2 \left(\frac{1}{p_i} \right) = - \sum_{i=1}^M p_i \log_2 p_i$$

Dove con M si indica il numero di elementi nell'insieme X . Questa formula si riassume nel seguente script, dove la variabile contenente l'immagine in bianco e nero viene trasposta e convertita in un vettore monodimensionale. In secondo luogo attraverso la funzione `numpy.histogram` vengono contate il numero di occorrenze per ogni valore di pixel. Successivamente, per calcolare la probabilità, si divide il numero di occorrenze per il numero totale di occorrenze, escludendo eventuali valori diversi da zero. Una volta calcolare la probabilità, attraverso le funzioni `numpy.sum` e `numpy.log2` si ottiene il valore dell'entropia $H(X)$.

```

1 | # flatten the transposed matrix to read pixels row by row
2 | raster_scan = np.transpose(gray_img).flatten()
3 |
4 | # count the occurrences of each pixel value
5 | occurrences = np.histogram(raster_scan, bins=range(256))[0]
6 |

```

```

7      # calculate the relative frequencies
8      rel_freq = occurrences / np.sum(occurrences)
9
10     # remove zero-values of probability
11     p = rel_freq[rel_freq > 0]
12
13     # compute the entropy
14     entropy_x = - np.sum(p * np.log2(p))

```

Dopo aver eseguito lo script, l'entropia dell'immagine scelta è di **7.581 bpp**.

1.3 Codifica con dizionario

La terza task chiede di utilizzare una compressione a dizionario, come **zip** nel caso di *Windows* per poi calcolare il *bitrate* risultante. Lo script necessario per soddisfare la richiesta è presentato nel frammento di codice sottostante. In particolare le prime righe si occupano di "zippare" il file mentre le istruzioni seguenti estraggono la dimensione dell'immagine compressa (in bytes). Infine nelle ultime linee del frammento di codice viene computato l'effettivo *bitrate* dividendo la dimensione del file compresso con la dimensione dell'immagine originale.

```

1      # change the current working directory to the directory containing
        the image
2      os.chdir(path_to_img)
3
4      # zip the image
5      cmd = f"zip {img_file_name}.zip {img_file_name}{img_extension}"
6      os.system(cmd)
7
8      # get the zip bytes
9      zip_bytes = os.stat(f"{img_file_name}.zip").st_size
10
11     # get img size
12     height, width = gray_img.shape
13     img_size = width * height
14
15     # get the birate
16     zip_bitrate = zip_bytes * 8 / img_size

```

Dunque, dopo aver eseguito lo script si ottiene il valore del bitrate, corrispondente a **1.329 bpp**.

1.4 Discussione risultati parziali

Osservando il valore di entropia ottenuto nel punto 1.2 si osserva che quest'ultima è molto vicina ad 8, indicando il fatto che l'immagine trovata, in ogni pixel, contiene 7.581 bit di informazione su un limite massimo di di 8 bpp. Il motivo per cui tale valore non è esattamente 8 può dipendere dal fatto che l'immagine (figura 1) contiene

zone molto uniformi rispetto ad altre, per esempio il cielo e i grattacieli, i quali sono tutti della stessa tonalità di grigio e sono più o meno uniformi rispetto al personaggio raffigurato.

Il fatto che la compressione attraverso la codifica con dizionario abbia un bitrate di 1.329 bpp (punto 1.3) sta a significare che, nonostante l'entropia dell'immagine fosse alta, la compressione è riuscita a sfruttare in maniera ottimale la ridondanza dei dati, precedentemente puntualizzata. Infatti si osserva che dividendo l'entropia iniziale con il bitrate ottenuto dalla codifica con dizionario si ottiene un **tasso di compressione** di 5.706, che indica una compressione particolarmente efficace, nonostante l'elevata entropia iniziale.

1.5 Codifica semplice

La quinta richiesta dell'homework è quella di effettuare una codifica predittiva *semplice*. In particolare, data l'immagine, definita come un vettore chiamato $x(n)$, la codifica predittiva è definita come segue:

$$x(n) = \begin{cases} 128 & \text{se } n = 0 \\ x(n-1) & \text{altrimenti} \end{cases}$$

Di conseguenza l'errore di predizione $y(n)$ sull'immagine $x(n)$ è dato da:

$$y(n) = \begin{cases} x(n) - 128 & \text{se } n = 0 \\ x(n) - x(n-1) & \text{altrimenti} \end{cases}$$

Tale codifica si traduce nel seguente codice Python, dove viene utilizzato il vettore `raster_scan`, calcolato nel punto 1.2, contenente il vettore dell'immagine in scala di grigi.

```

1 | # calculate prediction error
2 | simple_coding_error = np.concatenate(([raster_scan[0] - 128],
   | np.diff(raster_scan.astype(float))))
3 |
4 | # plot error graph
5 | simple_coding_error_img =
   | np.transpose(np.reshape(np.abs(simple_coding_error),
   | (width, height)))

```

Dopo aver eseguito lo script soprastante si ottiene l'immagine mostrata nella figura 2. Dall'immagine si può notare che le zone più chiare, ovvero le zone in cui il predittore ha fatto più errori, sono le zone dei contorni, in particolare del personaggio raffigurato. Questo perché la differenza dei colori tra una sezione e l'altra è particolarmente accentuata. D'altro canto, le zone di colore blu scuro sono quelle in cui i colori sono più uniformi: i palazzi e il cielo sono quasi dello stesso colore, suggerendo una zona dove la variazione di colore - e quindi di informazione - è molto bassa.

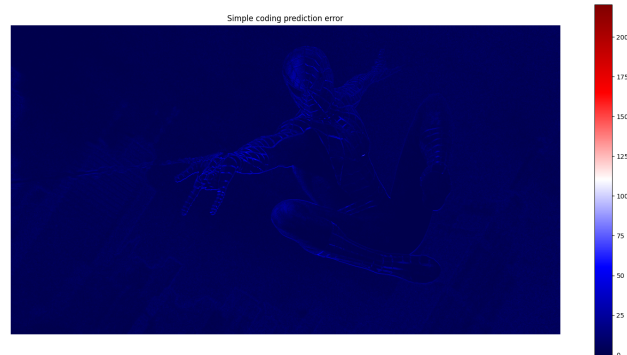


Figura 2: Rappresentazione del modulo dell'errore di predizione nella codifica semplice.

1.6 Entropia dell'errore di predizione

La richiesta successiva è quella di calcolare il valore dell'entropia dell'errore di predizione $y(n)$. Utilizzando uno script del tutto simile a quello utilizzato nella task 1.2 è quindi possibile calcolare l'entropia richiesta.

```

1      # count the occurrences of each prediction error value
2      occ, _ = np.histogram(simple_coding_error, bins = range(-255,
3                               256))
4
5      # calculate the relative frequencies and remove any probability == 0
6      freqRel = occ / np.sum(occ)
7      p = freqRel[freqRel > 0]
8
9      # calculate the entropy
10     entropy_y = - np.sum(p * np.log2(p))

```

Dopo l'esecuzione dello script si ottiene che l'entropia dell'errore di predizione semplice è **4.382 bpp**.

1.7 Exp Golomb

Seguendo le direttive della task numero 7, calcoliamo la codifica esponenziale di Golomb (detta anche *exp-Golomb*), la quale dato un numero intero ritorna la sua codifica. Quest'ultima è più efficace rispetto ad una normale codifica binaria in quanto durante la codifica vengono risparmiati diversi bit, migliorando la compressione.

Tale codifica può essere esaminata prima affrontando il caso in cui tutti i numeri da codificare siano interi positivi e poi tale concetto si può generalizzare. La codifica di Golomb per interi positivi, detta anche *exponential Golomb unsigned*, è definita come segue.

capisci cosa
significa il
valore EG-
bpp

$$\text{eg_unsigned}(n) = \begin{cases} 1 & \text{se } n = 0 \\ \text{zeros}(\lfloor \log_2(n+1) \rfloor) + \text{dec2bin}(n+1) & \text{altrimenti} \end{cases}$$

Dove `zeros(k)` indica una funzione che ritorna una stringa di k zeri mentre la funzione `dec2bin(n)` ritorna il valore binario del numero naturale n . A questo, dopo aver definito la codifica senza segno, la codifica con segno, detta anche *exponential Golomb signed*, è immediata:

$$\text{eg_signed}(n) = \begin{cases} \text{eg_unsigned}(2n-1) & \text{se } n > 0 \\ \text{eg_unsigned}(-2n) & \text{altrimenti} \end{cases}$$

Dopo aver definito matematicamente la codifica, è possibile tradurla in due funzioni Python molto seplci, che riassumono esattamente quanto già osservato.

```

1  def exp_golomb_signed(n: int) -> str:
2      # Computes the Exp-Golomb code for signed integers
3
4      if n > 0:
5          return exp_golomb_unsigned(2 * n - 1)
6
7      return exp_golomb_unsigned(-2 * n)
8
9
10 def exp_golomb_unsigned(n: int) -> str:
11     # Computes the Exp-Golomb code for non-negative integers
12
13     # handle the case where N is zero
14     if n == 0:
15         return '1'
16
17     # returns the coded string of bits
18     return '0' * int( math.floor( math.log2(n + 1) ) ) +
        format(n + 1, 'b')
```

Possiamo quindi occuparci di calcolare quanto richiesto dalla consegna. Per calcolare il numero di bit necessari utilizzando la codifica esponenziale di Golomb, è sufficiente calcolare la codifica per ogni errore della predizione semplice. La lunghezza delle codifica di ciascun valore viene sommata ottenendo quindi il numero totale di bit necessari per codificare l'errore. Infine, per ottenere il bitrate di codifica è necessario dividere il numero totali di bit per la grandezza dell'immagine. Il seguente script ricopre esattamente queste direttive.

```

1  def exp_golomb_count(vector: list) -> int:
2      # Calculate the bitrate based on the exponential Golomb code for
3      # a given vector.
4
5      bit_count = 0
6      for symbol in vector:
```

```

6         bit_count += len(exp_golomb_signed(int(symbol)))
7
8         return bit_count
9
10    exp_golomb_bit = exp_golomb_count(simple_coding_error)
11
12    exp_golomb_bpp = exp_golomb_bit / img_size

```

Dopo l'esecuzione dello script, il numero di bit necessari per la codifica è 41305140 mentre il bitrate di codifica ha il valore di **4.980 bpp**.

1.8 Conclusioni

Cerchiamo ora di ottenere dei dati statistici che ci permettano di trarre delle informazioni utili. Si sono quindi prese le immagini in scala di grigi - formato **.pgm** - fornite nella demo, e si è eseguito il codice per ciascuna immagine. I risultati trovati eseguendo lo script descritto nelle precedenti sezioni per ogni immagine sono stati riassunti nella seguente tabella (1).

img	.pgm	.zip	error	EG-coding
einst	6.785	6.391	5.343	6.897
house	7.056	4.002	3.613	3.786
lake	7.484	6.884	5.656	7.094
lena	7.445	6.808	4.675	5.542
peppers	7.594	7.081	5.026	6.261
plane	6.704	5.714	4.675	5.252
spring	7.157	6.918	5.342	6.757

Tabella 1: Dati ottenute dalle varie istanze di esecuzione dello script, con immagini diverse.

Osservando la tabella riassuntiva si possono notare dei pattern tra i vari risultati. Prima di tutto va osservato che, ragionevolmente, l'immagine originale ha un'entropia maggiore rispetto ad una qualsiasi forma di codifica o di compressione. Questo risultato suggerisce che tale ridondanza può essere sfruttata al fine di comprimere l'immagine. Osservando infatti la seconda colonna, contenete i risultati della codifica a dizionario, possiamo osservare che tali valori sono nettamente ridotti rispetto all'immagine originale; in particolare si osserva una differenza notevole nell'immagine **house**.

In secondo luogo si osserva la differenza tra il bitrate dell'errore della codifica semplice e quello della compressione in zip. Si nota che il primo, in ciascuna istanza di esecuzione è minore del secondo. Questo probabilmente è dovuto al fatto che la codifica predittiva riesce a comprimere meglio il file rispetto alla codifica generica.

forse ho detto una cazzata

2 Codifica avanzata

2.1 Codifica avanzata

La penultima richiesta, esige di creare un nuovo tipo di codifica, detta *avanzata* definita come segue:

$$y(n, m) = \begin{cases} x(n, m) - 128 & \text{se } n = m = 0 \\ x(n, m - 1) & \text{se } n = 0 \\ x(n - 1, m) & \text{se } m = 0 \\ \text{med}[x(n, m - 1), x(n - 1, m), x(n - 1, m - 1)] & \text{se } m = \text{MAX} \\ \text{med}[x(n, m - 1), x(n - 1, m), x(n - 1, m + 1)] & \text{altrimenti} \end{cases}$$

Dove in questo caso l'immagine anziché essere vista come un vettore x è vista come una matrice di dimensione $n \times m$. Le richieste della codifica sono tradotte in uno script contenente due cicli annidati - il primo che itera lungo le righe della matrice e il secondo che itera lungo le colonne - dove all'interno sono presenti una serie di `if` statement che soddisfano la definizione di codifica avanzata precedentemente citata. Inoltre si osserva la mediana dei tre valori è calcolata tramite una funzione definita dall'utente, descritta anch'essa all'interno dello script.

```
1  # median function for advanced coding
2  def median(a, b, c):
3      vector = [a, b, c]
4      vector.remove(min(vector))
5      vector.remove(max(vector))
6
7      return vector[0]
8
9
10
11  # blank image
12  predicted_img = np.zeros_like(gray_img)
13
14  # iterates through the rows (height)
15  for row in range(height):
16
17      # iterates through the cols (width)
18      for col in range(width - 1):
19
20          if row == 0 and col == 0: # first pixel
21              predicted_img[row][col] = gray_img[row][col] - 128
22
23          elif row == 1:             # first row
24              predicted_img[row][col] = gray_img[row][col - 1]
25
26          elif col == 1:             # first col
27              predicted_img[row][col] = gray_img[row - 1][col]
```

```

28
29     elif col == (width - 1): # last col
30         predicted_img[row][col] = median(gray_img[row -
31                                           1][col], gray_img[row][col - 1], gray_img[row -
32                                           1][col - 1])
33
34     else:
35         # other cases
36         predicted_img[row][col] = median(gray_img[row -
37                                           1][col], gray_img[row][col - 1], gray_img[row -
38                                           1][col + 1])
39
40 adv_coding_error = gray_img - predicted_img

```

Dopo aver eseguito lo script è possibile mostrare l'errore di predizione sottraendo l'immagine predetta `predicted_img` con l'immagine iniziale `img` e "plottando" il suo valore assoluto come segue mediante uno script simile a quello mostrato per la codifica semplice (1.5). L'immagine mostrata in figura 3 è ciò che risulta degli errori commessi durante la predizione.



Figura 3: Rappresentazione del modulo dell'errore di predizione nella codifica avanzata.

Si osserva che per quanto l'immagine 3 sia simile alla figura 2, è presente una differenza: è sufficiente mostrare un'immagine che contenga la differenza in modulo tra le variabili `adv_coding_error` e `simple_coding_error`. Dal risultato, mostrato nella figura 4, si può notare che non è di colore uniforme ma presenta diverse sfumature di blu, suggerendo quindi una bassa differenza tra le due che quindi le rende non identiche.

Tutti i valori della codifica avanzata sono essenzialmente corretti in quanto sono minori rispetto alla codifica semplice

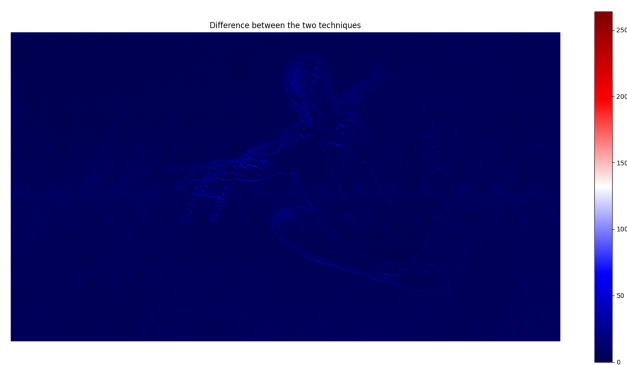


Figura 4: Rappresentazione della differenza tra l'errore della codifica semplice e della codifica avanzata.