Práctica 2: Código Huffman

Guillermo Martín Sánchez

26/02/20

1 Introducción

El objetivo de esta práctica es la construcción de los códigos de Huffman de las variables aleatorias $S_{English}$ y $S_{Spanish}$ para la codificación y decodificación de palabras de estos idiomas a binario. También estudiamos el Primer Teorema de Shannon y la eficiencia del código de Huffman frente a otros códigos binarios.

2 Material usado

2.1 Ejercicio 1

Hallar el código Huffman binario de $S_{English}$ y $S_{Spanish}$, sus longitudes medias $L(S_{English})$ y $L(S_{Spanish})$, y comprueba que se satisface el Primer Teorema de Shannon.

Para la primera parte, hemos utilizado el código facilitado para construir, de manera iterativa y como se ha visto en clase, el árbol de Huffman para las dos variables aleatorias en cuestión. A partir de ahí, recorriendo el árbol adecuadamente hemos construido un diccionario que a cada posible estado de la variable aleatoria le asignaba una correspondiente cadena binaria: el código Huffman binario de cada carácter.

Para calcular la longitudes medias hemos usado la fórmula

$$L(C) := \frac{1}{W} \sum_{i=1}^{N} w_i |c_i| \tag{1}$$

donde $|c_i|$ es la longitud de cada cadena binaria, w_i la frecuencia del carácter codificado por c_i y $W = \sum_{i=1}^{N} w_i$.
Por otro lado hemos calculado la Entropía de Shannon de las dos variables como

$$H(C) = -\sum_{j=1}^{N} P_j log_2(P_j) \tag{2}$$

donde P_j es la probabilidad (en este caso la aproximamos con la frecuencia relativa) de cada estado. Terminamos comprobando que el Primer Teorema de Shannon ($H(C) \leq L(C) < H(C) + 1$) se cumple.

2.2 Ejercicio 2

Codificar con dicho código la palabra "fractal" en ambas lenguas y comprobar la eficiencia de longitud comparada con el código binario usual

Usamos las codificaciones anteriores para, carácter a carácter, codificar la palabra en código de Huffman binario. Luego calculamos cuánto ocuparía en un sistema binario usual. Para ello calculamos cuántos bits harían falta para codificar cada estado de la variable aleatoria S con la fórmula $nbits = \lceil log_2(|S|) \rceil$ y multiplicamos $nbits$ por el número de caracteres que tiene la palabra.

2.3 Ejercicio 3

Decodifica la siguiente palabra del inglés:"10101000011110111111100"

Hemos obtenido el diccionario inverso del código de Huffman y lo hemos usado para recorrer la palabra decodificándola.

3 Resultados y discusión

3.1 Ejercicio 1

Para $S_{English}$ hemos obtenido que $H(S_{English}) = 4.315$ y $L(S_{English}) = 4.342$. Análogamente con $S_{Spanish}$ obtenemos $H(S_{Spanish}) = 4.184$ y $L(S_{Spanish}) = 4.214$. Por lo tanto en ambos casos se cumple el Primer Teorema de Shannon ya que $4.315 \leq 4.342 < 5.315$ y $4.184 \leq 4.214 < 5.184$. De hecho vemos que la codificación es muy eficiente en ambos casos ya que $L(C)$ y $H(C)$ están muy cercanos.

3.2 Ejercicio 2

Hemos obtenido que la codificación de Huffman de la palabra "fractal" desde el inglés es "0110001001011100011110101111110". Su longitud es de 31 bits mientras que en un sistema binario usual hemos calculado que sería de 42 bits. Como era de esperar $31 \leq 42$.

De igual modo, en español hemos obtenido la codificación "01101000110001110111110000010" de longitud 30 (≤ 42).

3.3 Ejercicio 3

Hemos obtenido que la codificación de Huffman de la palabra "10101000011110111111100"" del inglés es "henal".

4 Conclusión

El cálculo del código Huffman binario de una variable aleatoria es un proceso directo de realizar que nos genera un método de codificación eficaz para palabras formadas por los posibles estados de estas variables. Además, hemos visto que los resultados teóricos (Primer Teorema de Shannon y eficiencia de la codificación) se cumplen en estos casos concretos.

5 Código

```
"""
Coursework 2: Huffman Codification
"""

import os
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from collections import defaultdict

ubica = "C:/Users/guill/Documents/Carrera/GEOComp"

os.getcwd()
os.chdir(ubica)


with open('auxiliar_en_pract2.txt', 'r') as file:
      en = file.read()

with open('auxiliar_es_pract2.txt', 'r') as file:
      es = file.read()


en = en.lower()
es = es.lower()

# Count frequency in each text
from collections import Counter
tab_en = Counter(en)
```

```
tab_es = Counter(es)


# Construct DataFrame and sort
tab_en_states = np.array(list(tab_en))
tab_en_weights = np.array(list(tab_en.values()))
tab_en_probab = tab_en_weights/float(np.sum(tab_en_weights))
distr_en = pd.DataFrame({'states': tab_en_states, 'probab': tab_en_probab})
distr_en = distr_en.sort_values(by='probab', ascending=True)
distr_en.index=np.arange(0,len(tab_en_states))

tab_es_states = np.array(list(tab_es))
tab_es_weights = np.array(list(tab_es.values()))
tab_es_probab = tab_es_weights/float(np.sum(tab_es_weights))
distr_es = pd.DataFrame({'states': tab_es_states, 'probab': tab_es_probab })
distr_es = distr_es.sort_values(by='probab', ascending=True)
distr_es.index=np.arange(0,len(tab_es_states))

# Given a frequency table 'distr', computes a branch of the Huffman tree
def huffman_branch(distr):
    states = np.array(distr['states'])
    probab = np.array(distr['probab'])
    state_new = np.array([''.join(states[[0,1]])])
    probab_new = np.array([np.sum(probab[[0,1]])])
    codigo = np.array([{states[0]: 0, states[1]: 1}])
    states =  np.concatenate((states[np.arange(2,len(states))], state_new), axis=0)
    probab =  np.concatenate((probab[np.arange(2,len(probab))], probab_new), axis=0)
    distr = pd.DataFrame({'states': states, 'probab': probab, })
    distr = distr.sort_values(by='probab', ascending=True)
    distr.index=np.arange(0,len(states))
    branch = {'distr':distr, 'codigo':codigo}
    return(branch)

# Given a frequency table 'distr', computes the Huffman tree
def huffman_tree(distr):
    tree = np.array([])
    while len(distr) > 1:
        branch = huffman_branch(distr)
        distr = branch['distr']
        code = np.array([branch['codigo']])
        tree = np.concatenate((tree, code), axis=None)
    return(tree)

# Given a Huffman 'tree', computes it's Huffman table
def huffman_code(tree):
    dict = defaultdict(str)
```

```
    for i in range(len(tree) - 1, -1, -1):
        word0 = list(tree[i].keys())[0];
        for c in word0:
            dict[c]+='0'
        word1 = list(tree[i].keys())[1];
        for c in word1:
            dict[c]+='1'
    return(dict)

# Given a frequency table 'distr' and a Huffman table 'code', computes the
# average length L = 1/W * sum(wi * |ci|)
def average_length(distr, code):
    length = 0
    for i,row in distr.iterrows():
        length+=row['probab']*len(code[row['states']])
    return length

# Given a word 'word', and a Huffman table 'code' with 'word' belonging to the
# system 'code' codifies, computes the minimum number of bits necessary to
# represent the word, supossing we use the same number of bits to encode each
# possible state
def binary_length(word,code):
    return int(np.ceil(np.log2(len(code)))*len(word))

# Given a word 'word', and a Huffman table 'code' with 'word' belonging to the
# system 'code' codifies, computes the Huffman codification of that word
def codify(word,code):
    cod = ''
    for a in word:
        cod += code[a]
    return cod

# Given an injective dictionary 'code', computes its inverse dictionary
def inverse_dict(code):
    invcode = {}
    for x in code:
        invcode[code[x]] = x
    return invcode

# Given a word 'word' in binary, and the inverse of a Huffman table 'invcode',
# computes the corresponding word in the system that the Huffman table codifies
def decodify(word,invcode):
    acum = ''
    decod = ''
    for b in word:
        acum += b
```

```
        if(acum in invcode):
            decod += invcode[acum]
            acum = ''
    return decod

# Given a frequency table 'distr', computes its cumulative variable
def cumulative_variable(distr):
    acum = 0
    y = []
    for i,row in distr.iterrows():
        acum +=row['probab']
        y.append(acum)
    return y

# Given a cumulative variable 'y', computes its Gini Index
def gini_index(y):
    acum = 0
    for j in range(len(y)-1):
        acum += (y[j] + y[j+1])
    return 1 - (1/len(y))*acum

# Given a frequency table 'distr', computes its Hill Index with q = 2
def D2_Hill(distr):
    acum = 0
    for i,row in distr.iterrows():
        acum +=row['probab']**2
    return 1/acum

# Given a frequency table 'distr', computes its Entropy
def entropy(distr):
    entr = 0
    for i,row in distr.iterrows():
        entr+=row['probab']*np.log2(row['probab'])
    return -entr

#%%
"""
Exercise 1: Huffman Code
"""

# English
# Construct the Huffman tree
tree_en = huffman_tree(distr_en)

# Construct the Huffman table
code_en = huffman_code(tree_en)
```

```
# Compute the average length of the Huffman codification
l_en = average_length(distr_en,code_en)

# Compute the entropy of English
h_en = entropy(distr_en)

print("The average length of the random variable SEnglish is {:.4g}".format(l_en))

# Check Shannon's First Theorem: H(C) <= L(C) < H(C) + 1
print("Shannon's First Theorem: {:.4g} <= {:.4g} < {:.4g}".format(h_en, l_en, h_en+1))


# Spanish
tree_es = huffman_tree(distr_es)
code_es = huffman_code(tree_es)
l_es = average_length(distr_es,code_es)
h_es = entropy(distr_es)
print("The average length of the random variable SSpanish is {:.4g}".format(l_es))
print("Shannon's First Theorem: {:.4g} <= {:.4g} < {:.4g}".format(h_es, l_es, h_es+1))

#%%
"""
Exercise 2: Codification
"""
# English

# Codify with the Huffman codification
cod_en = codify('fractal',code_en)

# Estimate number of bits in usual binary codification
binlen_en = binary_length('fractal',code_en)

# Print in Huffman code
print("Codification Huffman: ", cod_en)

# Check the Huffman codification is shorter than usual binary
print("Length in Huffman codification:", len(cod_en),
      "< Length in Binary codification:", binlen_en)

# Spanish
cod_es = codify('fractal',code_es)
binlen_es = binary_length('fractal',code_es)
print("Codification Huffman: ", cod_es)
print("Length in Huffman codification:", len(cod_es),
      "< Length in Binary codification:", binlen_es)
```

```
#%%
"""
Exercise 3: Decodification
"""
# Construct inverse dictionary for decodification
invcode_en = inverse_dict(code_en)

# Decodify Huffman code
decod_en = decodify('1010100001111011111100', invcode_en)

# Print decodified word
print("The word 1010100001111011111100 =", decod_en)

#%%
"""
Exercise 4: Gini index and Hill index
"""
x = np.linspace(0,1,len(distr_en))

# Compute cumulative variable
y = cumulative_variable(distr_en)

# Print Lorentz curve
plt.text(0.6, 0.3, 'S', fontsize=24)
plt.text(0.85, 0.1, 'A', fontsize=24)
plt.plot(x,y)
ax = plt.gca()
ax.set_xlim([0,1])
ax.set_ylim([0,1])
plt.fill_between(x,0,x)
plt.fill_between(x,0,y)
plt.xlabel('Cumulative frequency (x)')
plt.ylabel('Cumulative variable (y)')

# Compute and print Gini Index
g = gini_index(y)
print("The Gini Index of English is G = {:.4g}".format(g))

# Compute and print Hill(q=2) Index
d = D2_Hill(distr_en)
print("The Hill Index for q = 2 of English is 2D = {:.4g}".format(d))
```