Práctica 1 GCOMP: Código Huffman y 1er Teorema de Shannon

Enrique Carro Garrido

I. Introducción (motivación/objetivo de la práctica)

Uno de los pilares fundamentales de la Geometría Computacional es la aplicación de conocimientos geométricos en sistemas discretos de información.

Los datos procesados para conformar una información se extraen de fuentes que suelen presentar una elevada complejidad en su modelización matemática. De hecho, si nuestros datos proceden de un sistema físico, lo más probable es que se trate de un sistema dinámico, e incluso no lineal.

Dentro de estos sistemas dinámicos, en esta práctica nos centramos en conjuntos finitos para abordar así las medidas geométricas de la información y la codificación, en particular, la entropía, que describe sistemas dinámicos desde el punto de vista estático, centrándose en todo el espacio de configuración de los estados.

La entropía es una medida extensiva que describe la complejidad (o capacidad de albergar información) de un sistema, a través de la diversidad y disparidad de aparición de cada fase. Esta diversidad de información es clave a la hora de diseñar una codificación.

En esta práctica se pretende profundizar en la Codificación de Huffman, íntimamente relacionada con la entropía de la información, abordando dos propiedades importantes de la misma:

a. Satisface el 1^{er} Teorema de Shannon, es decir, el algoritmo de Huffman acota la longitud media del código simbólico según: $H(C) \leq L(C) < H(C) + 1$.

b. La codificación es más eficiente que la binaria trival, ya que la longitud de la cadena de Huffman es menor que la trivial.

II. Material usado (método y datos)

a. El **método** que se ha utilizado es la *Codificación de Huffman*, una codificación binaria prefijo (sin encadenar prefijos repetidos de longitud variable) empleada para analizar probabilidades de los carácteres de los lenguajes S_{Eng} (alfabeto del inglés) y S_{Esp} (alfabeto del español).

b. Los **datos** utilizados para su desarrollo son:

- *"GCOM2023_pract1_auxiliar_eng.txt"* con la muestra en inglés.
- *"GCOM2023_pract1_auxiliar_eng.txt"* con la muestra en inglés.
- *"GCOM2023practica1_plantilla.py"* con la plantilla subida en el campus virtual que incluye la creación del árbol de Huffman.

III. Resultados

Las respuestas a cada apartado que se muestran por consola al ejecutar el código que aparece en el anexo se resumen a:

a. *A partir de las muestras dadas, hallar el código Huffman binario de SEng y SEsp, y sus longitudes medias $L(SEng)$ y $L(SEsp)$. Comprobar que se satisface el Primer Teorema de Shannon.*

Para *Seng* se comprueba que:

- $L(Seng) = 4{,}436974789915966$.
- $H(Seng) = 4{,}338300577316106$.

- Por lo tanto, como $H(Seng) <= L(Seng) < H(Seng) + 1$, se satisface el Teorema de Shannon para *Seng*.

Para *Sesp* se comprueba que:

- $L(Seng) = 4{,}436974789915966$.
- $H(Seng) = 4{,}338300577316106$.
- Por lo tanto, como $H(Seng) <= L(Seng) < H(Seng) + 1$, se satisface el Teorema de Shannon para *Seng*.

En este documento no he incluido

b. *Utilizando los códigos obtenidos en el apartado anterior, codificar la palabra cognada X = "dimension" para ambas lenguas. Comprobar la eficiencia de longitud frente al código binario.*

Para *Seng*:

- Codificación en inglés: 00111010101110000100001000101011010000.
- Longitud del código en inglés: 37.
- Longitud de la codificación usual: 54.
- La codificación de Huffman en *Seng* es más eficiente que la usual.

Para *Sesp*:

- Codificación en español: 11110001101100000100110100011100011001.
- Longitud del código en español: 37.
- Longitud de la codificación usual: 54.
- La codificación de Huffman en *Sesp* es más eficiente que la usual.

c. *Decodificar la siguiente palabra del inglés: "0101010001100111001101111000101111110101010001110".*

La palabra decodificada es "isomorphism".

IV. Conclusión

Gracias a los resultados obtenidos en esta práctica se puede concluir que las dos propiedades de la Codificación de Huffman que se describen en la introducción se verifican para los textos que se mencionan en la sección de método y datos usados.

V. Anexo con el código utilizado

```
"""
Enrique Carro Garrido

Práctica 1. Código de Huffmann y Teorema de Shannon
"""

import os
import numpy as np
import pandas as pd

# Vamos al directorio de trabajo
os.getcwd()

with open('GCOM2023_pract1_auxiliar_eng.txt', 'r',encoding="utf8") as file:
      en = file.read()

with open('GCOM2023_pract1_auxiliar_esp.txt', 'r',encoding="utf8") as file:
      es = file.read()

# Contamos cuantos caracteres hay en cada texto
```

```
from collections import Counter
tab_en = Counter(en)
tab_es = Counter(es)

# Transformamos en formato array de los caracteres (states) y su frecuencia
# Finalmente realizamos un DataFrame con Pandas y ordenamos con '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))

## Ahora definimos una función que haga exáctamente lo mismo
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)

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)

# Respuestas

# i) A partir de las muestras dadas, hallar el código Huffman
# binario de SEng y SEsp, y sus longitudes medias L(SEng) y L(SEsp).
# Comprobar que se satisface el Primer Teorema de Shannon.

# Hallar el código Huffman binario de SEng y SEsp. Para ello
# utilizamos las siguientes funciones

def build_dictionary(tree):
    """
    Args:
        tree (np.array): Lista de ramas del árbol de huffman,
            en el último elemento se encuentran las dos primeras
            ramas de ese árbol.

    Return:
```

```
        coder (dict): Diccionario con clave = caracter del texto (estado)
            y valor = código simbólico binario correspondiente.
    """

    coder = dict()
    # Recorremos cada elemento del árbol empezando por abajo
    for branch in tree:
        # Recorremos cada rama de ese elemento
        for i in range(2):
            # items = cadena correspondiente a esa rama
            items = list(branch.items())[i]
            # Si items es un solo caracter entonces inicializamos esa entrada del diccionario
            if len(items[0]) <= 1:
                coder[items[0]] = str(items[1])
            # Si no modificamos la entrada añadiéndole el bit correspondiente a esa rama por la izquier
            else:
                for state in items[0]:
                    coder[state] = str(items[1]) + coder[state]
    return coder



def code_from_text(coder, text):
    """
    Args:
        coder (dict): Diccionario con clave = caracter del texto (estado)
            y valor = código simbólico binario correspondiente.
        text (str): Texto que se quiere codificar.

    Returns:
        code (str): Código simbólico de "text".
    """
    code = ''
    for state in text:
        code = code + coder[state]
    return code

# Código binario, longitud media y entropía de Seng
distr = distr_en
tree_en = huffman_tree(distr_en)
coder_en = build_dictionary(tree_en)
code_en = code_from_text(coder_en, en)
L_Seng = np.sum([len(coder_en[tab_en_states[i]]) * tab_en_probab[i] for i in range(len(tab_en_states))]
H_Seng = -np.sum([P * np.log2(P) for P in tab_en_probab])

# Código binario, longitud media y entropía de Sesp
distr = distr_es
tree_es = huffman_tree(distr_es)
coder_es = build_dictionary(tree_es)
code_es = code_from_text(coder_en, en)
L_Sesp = np.sum([len(coder_es[tab_es_states[i]]) * tab_es_probab[i] for i in range(len(tab_es_states))]
H_Sesp = -np.sum([P * np.log2(P) for P in tab_es_probab])

# Resultados:
print("-------------------------------------------------------------------------------")
print("                                APARTADO 1                                     ")
print("-------------------------------------------------------------------------------")
print()

# Escribimos los resultados para Seng y comprobamos que se satisface el Primer Teorema de Shannon
print("Texto Seng: " + en)
```

```
print("Código: "+ code_en)
print("L(Seng) = " + str(L_Seng))
print("H_Seng = " + str(H_Seng))
if H_Seng <= L_Seng and L_Seng < H_Seng + 1:
    print("Se satisface el Teorema de Shannon para el código simbólico de Seng")
else :
    print("No se satisface el Teorema de Shannon para el código simbólico de Seng")
print()

# Escribimos los resultados para Sesp y comprobamos que se satisface el Primer Teorema de Shannon en Se
print("Texto Sesp: " + es)
print("Código: "+ code_es)
print("L(Sesp) = " + str(L_Sesp))
print("H_Sesp = " + str(H_Sesp))
if H_Sesp <= L_Sesp and L_Sesp < H_Sesp + 1:
    print("Se satisface el Teorema de Shannon para el código simbólico de Sesp")
else :
    print("No se satisface el Teorema de Shannon para el código simbólico de Sesp")
print()
print()

# ii) Utilizando los códigos obtenidos en el apartado anterior, codificar la palabra cognada X = "dimen
# para ambas lenguas. Comprobar la eficiencia de longitud frente al código binario usual

X = "dimension"

# Codificación de Huffman y usual en inglés
code_X_eng = code_from_text(coder_en, X)
len_code_X_bin_eng = int(len(X) * np.ceil(np.log2(len(tab_en))))

# Codificación de Huffman y usual en español
code_X_esp = code_from_text(coder_es, X)
len_code_X_bin_esp = int(len(X) * np.ceil(np.log2(len(tab_es))))

print("-------------------------------------------------------------------------------------------")
print("                                    APARTADO 2                                             ")
print("-------------------------------------------------------------------------------------------")
print()
print("X = " + X)
print()
print('Codificación en inglés: ' + code_X_eng)
print('Longitud del código en inglés: ' + str(len(code_X_eng)))
print('Longitud de la codificación usual: ' + str(len_code_X_bin_eng))
if len(code_X_eng) < len_code_X_bin_eng:
    print("La codificación de Huffman en Seng es más eficiente que la usual")
else:
    print("La codificación de Huffman en Seng no es más eficiente que la usual")
print()

print('Codificación en español: ' + code_X_esp)
print('Longitud del código en español: ' + str(len(code_X_esp)))
print('Longitud de la codificación usual: ' + str(len_code_X_bin_esp))
if len(code_X_esp) < len_code_X_bin_esp:
    print("La codificación de Huffman en Sesp es más eficiente que la usual")
else:
    print("La codificación de Huffman en Sesp no es más eficiente que la usual")
print()
print()

# iii) Decodificar la siguiente palabra del inglés:
# "0101010001100111001101111000101111110101010001110"
```

```python
def text_from_code(code, coder):
    """
    Args:
        code (str): Código simbólico que se pretende decodificar.
        coder (dict): Diccionario con clave = caracter del texto (estado)
            y valor = código simbólico binario correspondiente.

    Returns:
        text (str): Texto decodificada a partir de "code".
    """
    # decoder = diccionario "inverso" de coder, clave = código simbólico
    # y valor = caracter del texto (estado). Es posible invertir el
    # diccionario porque la codificación es biyectiva.
    decoder = {coder[key]:key for key in coder.keys()}
    # text = Texto decodificado de code.
    text = ''
    # next = siguiente caracter en código binario
    next = ''
    # Recorremos cada bit de code
    for bit in code:
        # Se lo añadimos a next
        next += bit
        # Si el código acumulado tiene una entrada en el diccionario decoder
        # entonces se acumula el valor en text y se reinicia next.
        if next in decoder.keys():
            text += decoder[next]
            next = ''
    return text

word = text_from_code("0101010001100111001101111000101111110101010001110", coder_en)

print("-----------------------------------------------------------------------------------")
print("                                   APARTADO 3                                      ")
print("-----------------------------------------------------------------------------------")
print()
print("Código: 0101010001100111001101111000101111110101010001110")
print("Palabra decodificada: " + word)
```