PRÁCTICA 1:

CÓDIGO DE HUFFMAN Y PRIMER TEOREMA DE SHANNON

Adrián Pérez Peinador

Introducción/motivación

La principal motivación de esta práctica es afianzar los conceptos de entropía y longitud media, así como entender y aplicar el algoritmo de Huffman a dos alfabetos dados. El árbol de Huffman se creará a partir de dos muestras dadas, una para cada uno de los alfabetos. Una vez calculada la entropía y la longitud media de cada uno de los alfabetos será tarea sencilla ver que se cumple el Primer Teorema de Shannon. Además, otra de las grandes motivaciones de esta práctica será comprobar la eficiencia de la codificación de Huffman con respecto a una codificación binaria normal. Por último, codificaremos y codificamos una serie de palabras que se nos presentan.

Material usado

En el código se hace uso de varias librerías de Python. En concreto se usó os (para abrir las muestras de cada alfabeto y poder tratarlas), numpy (para los arrays de numpy) y pandas (a la hora de trabajar con dataframes para construir el árbol de Huffman). Asimismo, se usaron las muestras de texto proporcionadas por el profesor.

Aparte de estos recursos externos, para la práctica se definieron varias funciones para realizar tareas específicas. Estas funciones son las siguientes.

- ***huffman_branch*:** A partir de un dataframe con los estados y su frecuencia relativa fusiona los dos estados con menor frecuencia.
- ***huffman_tree*:** Dado un dataframe con los estados y su frecuencia relativa, usa la función anterior para crear el árbol de Huffman completo.
- ***codifica_palabra*:** Dada una palabra y un diccionario, calcula el código Huffman binario de esa misma palabra. Ésto se hace buscando en el diccionario la palabra carácter a carácter y concatenando sus códigos.
- ***decodifica_palabra*:** A partir de un código y un diccionario devuelve la palabra la de la que viene ese código según el alfabeto representado por el diccionario en cuestión. Lo que hace esta función es buscar el prefijo del código que coincide con algún carácter. Una vez encontrado se sabe que solo le pertenece a este por la propiedad de prefijo variable.

Tras organizar las palabras del texto en un dataframe, cada una con su respectiva frecuencia se usan las funciones *huffman_branch* y *huffman_tree* para crear el árbol de Huffman. Posteriormente, éste se recorre una vez y se guarda en un diccionario toda la

información necesaria para el ejercicio. En nuestro caso el diccionario tiene como clave el carácter y como valor tanto su código en binario como la longitud del mismo.

Con estos datos se puede obtener rápidamente la longitud media de los carácteres multiplicando la longitud de cada carácter por su frecuencia relativa. Además, también se calcula rápidamente la entropía de los sistemas y se puede poner de manifiesto que se cumple el Primer Teorema de Shannon para los dos propuestos.

Para el segundo apartado se usa la función *codifica_palabra*, con la que se codifica tanto para inglés como para español. Por último, para el último apartado se usa la función *decodifica_palabra* que recupera la palabra original.

Resultados

A continuación se exponen los resultados obtenidos para cada apartado pedido.

Apartado 1:

Para el español se obtiene una longitud media de 4.3713 y una entropía de 4.3383. Por otro lado, para el inglés tenemos una longitud media de 4.4369 y una entropía de 4.4121. Por tanto podemos afirmar que para ambos casos se cumple el Primer Teorema de Shannon.

Apartado 2:

En el segundo apartado obtenemos las codificaciones de la palabra *dimension* para el inglés y el español.
En inglés: 00111010101110000100001000101011010000
En español: 11110001101100000100110100011100010001

En binario usual la longitud de cada carácter es la misma para todos los caracteres, por tanto ha de ser al menos el logaritmo en base 2 del número de caracteres distintos que se tengan. En inglés tenemos 67 caracteres mientras que en español hay que añadir 10 vocales acentuadas, ü, Ü, ñ, Ñ, ¡ y ¿, así que el diccionario en este caso tiene 83 elementos. Para ambos casos el código de un carácter ha de tener al menos longitud 7 (2^6 = 64 < 67 < 83 < 128 = 2^7), por lo que la longitud de la palabra *dimension* será en ambos casos al menos 63 bits (9*7), bastante mayor que la longitud obtenida por Huffman (37 bits). Comparando las longitudes medias por carácter también se puede ver la diferencia. Con el código Huffman tenemos una longitud media menor que 4.5 en ambos alfabetos mientras que con la codificación binaria usual tendríamos una longitud de 7 mínimo.

Apartado 3:

En el último apartado se obtiene que la palabra decodificada es *isomorphism*.

Conclusión

A raíz de la realización de la práctica hemos podido comprobar que la codificación de Huffman consigue sintetizar la información y reducir el número medio de bits que ocupa la codificación de un carácter. Además hemos verificado que esta codificación cumple el Primer Teorema de Shannon y hemos afianzado conceptos como la entropía.

Código

A continuación se anexa el código escrito para esta práctica, comentado para aclarar al lector lo que se hace en cada momento. Además se indica qué partes del código pertenecen a cada apartado tanto en comentarios en el propio código como por consola al ejecutar el mismo.

```
"""
Práctica 1. Código de Huffmann y Teorema de Shannon
Pablo Torre y Adrián Pérez
"""

import os
import numpy as np
import pandas as pd
from collections import Counter

#### 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
tab_en = Counter(en)
tab_es = Counter(es)

#### Transformamos en formato array de los carácteres (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))
```

```
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)

##### Para obtener una rama, fusionamos los dos states con menor frecuencia
''.join(distr_en['states'][[0,1]])
branch_en = huffman_branch(distr_en);
tree_en = huffman_tree(distr_en)

''.join(distr_es['states'][[0,1]])
branch_es = huffman_branch(distr_es);
tree_es = huffman_tree(distr_es)

### Creamos los diccionarios recorriendo los árboles solo una vez
### la clave es el carácter y el valor contiene el código del carácter y su longitud

dic_en = dict.fromkeys(tab_en_states, ['', 0])
n = len(tree_en)
for i in range(len(tree_en)):
    string0 = list(tree_en[n-i-1].items())[0][0]
    string1 = list(tree_en[n-i-1].items())[1][0]

    for j in string0:
        dic_en[j] = [dic_en[j][0] + '0', dic_en[j][1] + 1]
```

```
    for j in string1:
        dic_en[j] = [dic_en[j][0] + '1', dic_en[j][1] + 1]

dic_es = dict.fromkeys(tab_es_states, ['', 0])
n = len(tree_es)
for i in range(len(tree_es)):
    string0 = list(tree_es[n-i-1].items())[0][0]
    string1 = list(tree_es[n-i-1].items())[1][0]

    for j in string0:
        dic_es[j] = [dic_es[j][0] + '0', dic_es[j][1] + 1]

    for j in string1:
        dic_es[j] = [dic_es[j][0] + '1', dic_es[j][1] + 1]

# Apartado i
print("\n APARTADO 1 \n")


print("\nDiccionario ingles \n")
print(dic_en)
print("\nDiccionario español\n")
print(dic_es)


### Calculamos la longitud media y la entropia para verificar que se cumple el
#    Primer Teorema de Shannon
print("\nVerificación del 1er Tma de Shannon\n")

long_media_en = 0
for i in range(len(tab_en_states)):
    long_media_en += dic_en[tab_en_states[i]][1]*tab_en_probab[i]
entropia_en = 0
for i in range(len(tab_en_probab)):
    entropia_en += tab_en_probab[i]*np.log2(tab_en_probab[i])
entropia_en = 0 - entropia_en
print("Para el ingles:")
print("\nLongitd media =", long_media_en)
print("Entropía =", entropia_en)
print("\n" + str(entropia_en) + " <= " + str(long_media_en) + " < " + str(entropia_en + 1))
```

```
for i in range(len(tab_es_probab)):
    entropia_es += tab_es_probab[i]*np.log2(tab_es_probab[i])
entropia_es = 0 - entropia_es

print("\nPara el español:")
print("\nLongitd media =", long_media_es)
print("Entropía =", entropia_es)
print("\n" + str(entropia_es) + " <= " + str(long_media_es) + " < " + str(entropia_es + 1))


# Apartado ii
print("\n APARTADO 2")

### Definimos ahora una función para codificar una palabra a partir de un diccionario

def codifica_palabra(palabra, diccionario):
    resultado = ''
    for i in palabra:
        resultado += diccionario[i][0]
    return resultado

x_es = codifica_palabra('dimension', dic_es)
x_en = codifica_palabra('dimension', dic_en)
print("\n Codificación ingles:")
print(x_en)
print("\n Codificación español:")
print(x_es)

# COMPROBACIÓN DE EFICIENCIA EN LONGITUD
"""
En binario usual la longitud de cada carácter es igual, es decir el logaritmo en
base 2 del número de caracteres distintos del diccionario.
En S_Eng tenemos 67 y en S_Esp tenemos que añadir 10 vocales acentuadas, ü, Ü, ñ, Ñ,¡ y ¿
Así que S_Esp tiene 83 elementos
En ambos casos la palabra debe tener al menos longitud 7 (2^6 = 64 < 67 < 83 < 128 = 2^7)
La longitud de la palabra dimension será en ambos casos al menos 9*7 = 63 bits
"""
print("\nLongitud codificación usual: 63")
print("Longitud codificación inglés:", len(x_en))
print("Longitud codificación español:", len(x_es))
```

```
# Apartado iii
print("\n APARTADO 3 \n")
# Definimos ahora la función inversa, que decodifica una palabra a partir de
# su codificación y un diccionario

def decodifica_palabra(codigo, diccionario):
    resultado = ''
    marcador = 0

    while(marcador < len(codigo)):
        for i in diccionario:
            if(len(codigo) - marcador < diccionario[i][1]):
                continue
            if(diccionario[i][0] == codigo[marcador:(marcador+diccionario[i][1])]):
                resultado += i
                marcador += diccionario[i][1]
                if(marcador >= len(codigo)):
                    break
    return resultado

print("Palabra decodificada:",
      decodifica_palabra('0101010001100111001101111000101111110101010001110', dic_en))
```