PRÁCTICA 1

CÓDIGO DE HUFFMAN Y PRIMER TEOREMA DE SHANNON

Pablo Torre Piñana y Adrián Pérez Peinador

Introducción

El objetivo de esta práctica es, dados dos alfabetos diferentes y una muestra para cada uno, hallar sus códigos Huffman binarios además de sus longitudes medias. Una vez tengamos esto podremos ver de forma sencilla si se cumple el Primer Teorema de Shannon, así como comprobar la utilidad de esta codificación y su eficiencia de longitud frente al código binario usual. Finalmente se nos pide codificar y decodificar una serie de palabras, con lo que comprobamos empíricamente que la codificación funciona. Para hacer todo esto tendremos que tener claros los conceptos de entropía de un estado y de un sistema, así como el concepto de longitud media del código Huffman resultante de aplicar el algoritmo.

Material usado

Para nuestro código hemos utilizado las librerías de os (para abrir las muestras de cada alfabeto y poder tratarlas), numpy (pues usamos los arrays de numpy en alguna ocasión) y pandas (para trabajar con dataframes al construir el árbol de huffman).

En nuestro código definimos varias funciones, que cumplen un papel distinto en el desarrollo de la práctica. Las funciones definidas 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*:** A partir de 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, nos devuelve el código Huffman binario de esa misma palabra. Simplemente lo hacemos buscando en el diccionario la palabra carácter a carácter y concatenando sus códigos.
- ***decodifica_palabra*:** Dado un código y un diccionario devuelve la palabra a la que se refiere este código en el alfabeto representado por el diccionario en cuestión. Lo que hacemos en esta función es ver qué prefijo del código coincide con algún carácter. Una vez encontrado sabremos que solo le pertenece a este (prefijo variable).

Después de separar las palabras del texto y organizarlas en un dataframe, cada una con su respectiva frecuencia usamos las funciones *huffman_branch* y *huffman_tree* para crear el árbol de Huffman. Una vez creado el árbol, lo recorremos una vez y guardamos en un diccionario toda la información que vamos a necesitar en un futuro. En nuestro caso el diccionario tiene como clave el carácter y como valor tanto su código en binario como la longitud del mismo.

Con este diccionario podemos acceder rápidamente a la longitud de cada carácter y multiplicarlo por su frecuencia relativa para obtener la longitud media. Además, también calculamos la entropía de ambos sistemas y ponemos de manifiesto que se cumple el Primer Teorema de Shannon.

Para el apartado 2 usamos la función *codifica_palabra*, con la que conseguimos codificar tanto para inglés como para español. Por último, para el último apartado usamos la función "inversa" que hemos definido como *decodifica_palabra*.

Resultados

Para cada apartado hemos obtenido los resultados que se pedían, aparte de unos cuantos resultados intermedios que nos han facilitado la tarea.

Apartado 1:

En español:	Longitud media = 4.3713	Entropía = 4.3383
En inglés:	Longitud media = 4.4369	Entropía = 4.4121

En ambos casos se satisface el Primer Teorema de Shannon.

Apartado 2:

dimension (en): 00111010101110000100001000101011010000
dimension (es): 11110001101100000100110100011100001001

En binario usual la longitud de cada carácter es igual, es decir al menos 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), por lo que la longitud de la palabra *dimension* será en ambos casos al menos 9*7 = 63 bits, bastante mayor que la longitud obtenida por Huffman (37 bits).

Apartado 3:

Palabra decodificada: isomorphism

Conclusión

Como resultado de esta práctica hemos podido comprobar como la codificación de Huffman cumple el Primer Teorema de Shannon, así como la eficiencia de longitud de esta codificación frente al código binario usual. Además, calcular la longitud media y la entropía de un sistema concreto ha hecho que afiancemos mucho más esos conocimientos.

Código

Preparación de los datos y definición de la función *huffman_branch:*

```
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)
```

Definición de *huffman_tree* y creación de los diccionarios:

```
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]
```

Solución al primer apartado:

```
# 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))

long_media_es = 0
for i in range(len(tab_es_states)):
    long_media_es += dic_es[tab_es_states[i]][1]*tab_es_probab[i]
entropia_es = 0
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))
```

Solución al segundo apartado y definición de *codifica_palabra*:

```
# 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)
```

Solución al tercer apartado y definición de *decodifica_palabra*:

```
# 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))
```