Memoria Practica 2 – Codificación Huffman

Jorge del Valle Vázquez

1 Introducción

Trabajaremos la codificación Huffman como ejemplo de código de encriptación para así poder introducir conceptos como la entropía, en este caso relacionada con la información. En este caso se obtiene un código binario a partir de un texto en inglés y otro en castellano, pues es materia de uso común usar esta codificación en análisis de probabilidades de caracteres en el lenguaje, como podría ser también los estados de sistemas cuánticos.

2 Material usado

Las librerías empleadas son **pandas** para hacer uso de los *dataframes* y **collections** que nos permite contar el número de apariciones de cada carácter en los textos proporcionados *'GCOM2022_pract2_auxiliar_esp.txt'* y *'GCOM2022_pract2_auxiliar_esp.txt'.* En primer lugar, obtenemos un *dataframe* que asocia a cada estado o carácter el número de apariciones en el texto, que luego transformamos a frecuencias relativas ordenadas crecientemente.

En cuanto a la construcción del árbol de Hufmann, que nos proporcionará los códigos esperados, el proceso consiste en formar un nodo hoja para cada carácter y fusionar los nodos de menor frecuencia relativa para dar lugar al padre de los nodos en el árbol que consiste en la concatenación de los caracteres de los nodos hijos y como frecuencia su suma. Para la posterior codificación se le asigna 0 a la rama del hijo de menor frecuencia y 1 a la de mayor. Se procede hasta dar con un único nodo que será el nodo raíz del árbol binario.

Para codificar, hemos ido guardando los nodos en una lista teniendo así la raíz en la ultima posición. Recorriendo la lista en orden inverso tendríamos un pseudo-recorrido por niveles. Así desde la raíz a las hojas (que son los caracteres) concatenamos al código de cada carácter , inicialmente vacío, un 0 o un 1 dependiendo de si aparece en la cadena de caracteres del hijo izquierdo o derecho respectivamente, que recordamos hace referencia al nodo de mayor frecuencia relativa. Indicamos dos métodos, uno que proporciona un diccionario carácter-código, y otro que incorpora al *dataframe* una columna con el código.

Se desarrolla también el cálculo de L (longitud 'media' de la codificación) $L = \sum_{i=1}^{N} w_i \, |c_i|$, pues los w_i están normalizados en frecuencias, y H(entropía) $H = -\sum_{i=1}^{N} P_i \log_2 P_i$. También la *comprobación del primer teorema de Shannon* $H \leq L < H + 1$.

Para decodificar hacemos uso de la propiedad *prefijo* de la codificación Huffman, que indica que no se encadenan prefijos repetidos de longitud variable. Para visualizar el concepto se propone hacerse uso del *dataframe* '*dc_es_ord*' que está ordenado por el código. Esto permite recorrer el código a decodificar y si identificamos en el un prefijo *p* que se corresponde con el código de un carácter *c* significa que hemos dado con la decodificación *c* de *p* y procederíamos a seguir con el código posterior a *p.* Para facilitar el proceso comenzamos creando el diccionario inverso al anterior código-carácter.

3 Resultados

i. Obtenemos la codificación de S_eng y S_esp(que tiene más caracteres, 45 frente a 38), para comparar vemos algunas codificaciones de los caracteres de mayor frecuencia(que conlleva menor longitud en bits de código).

S_eng: ‘ ‘=00, ‘h‘=101, ‘t‘=011, ‘e‘=1110, ‘s‘=0101, ‘o‘=11111, ‘a‘=11010

S_esp: ‘ ‘=111, ‘e‘=010, ‘a‘=001, ‘n‘=1001, ‘s‘=1000, ‘o‘=0110, ‘t‘=0000

De los cálculos de L y H se obtiene:

S_eng: L= 4.158163265306123 y H= 4.117499394903037

S_esp: L= 4.431924882629108 y H= 4.3943938614799665

y es sencillo comprobar que se sigue el teorema de Shannon en ambos casos.

ii. En el segundo apartado codificamos la palabra *medieval* con las codificaciones obtenidas para inglés y español, obteniendo

S_eng= 1111010111111011011111100011101111010011010110111 0

S_esp= 1100010100001011001010011101010011010 1

Además comprobamos que es eficiente comparando con la codificación binaria usual con $\sum_{i=1}^{N} |c_i| \leq \mathrm{N}[\log_2 N]$.

iii. En el último apartado decodificamos al inglés “10111101101110110111011111” obteniendo la palabra hello

Hello=10111101101110110111011111

Para reflejar aquí el proceso de decodificación comenzando con la cadena entera el primer prefijo identificado es ‘101que equivale a ‘h’. Para continuar de lado el 101 y buscaríamos un nuevo prefijo en1110110111011011101111.

4 Conclusión

De lo trabajado en la práctica podemos concluir la longitud L y la entropía son menores para el inglés; pero eso no indica que la codificación de medieval tenga menos dígitos, pues en este caso es lo contrario. También permite la comprensión de conceptos de medida como L y lo mucho que se aproxima a la entropía H con esta codificación Huffman. Además resulta interesante trastear con los resultados obtenidos para profundizar la comprensión de conceptos nuevos como la propiedad prefijo.

5 Anexo: Código

```
"""
Práctica 2
"""

import os
import numpy as np
import pandas as pd
```

```
import math
#### Vamos al directorio de trabajo####
os.getcwd()
#os.chdir(ubica)
#files = os.listdir(ruta)

with open('GCOM2022_pract2_auxiliar_eng.txt', 'r',encoding="utf8") as file:
      en = file.read()

with open('GCOM2022_pract2_auxiliar_esp.txt', 'r',encoding="utf8") as file:
      es = file.read()


#### Contamos cuantas letras hay en cada texto
from collections import Counter
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))

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

### Es decir:
states = np.array(distr['states'])
probab = np.array(distr['probab'])
state_new = np.array([''.join(states[[0,1]])])   #Ojo con: state_new.ndim
probab_new = np.array([np.sum(probab[[0,1]])])   #Ojo con: probab_new.ndim
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))
```

```
#Creamos un diccionario
branch = {'distr':distr, 'codigo':codigo}

## 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)

distr = distr_en
tree = huffman_tree(distr)
tree[0].items()
tree[0].values()
#Buscar cada estado dentro de cada uno de los dos items
list(tree[0].items())[0][1] ## Esto proporciona un '0'
list(tree[0].items())[1][1] ## Esto proporciona un '1'

def calculate_codes(tree):
    codes=dict()
    for node in reversed(tree):##tre[::-1]
        for symbol in list(node.items())[0][0]:
            if codes.get(symbol) == None:
                codes[symbol] =  '0'
            else:
                codes[symbol] +=  '0'
        for symbol in list(node.items())[1][0]:
            if codes.get(symbol) == None:
                codes[symbol] =  '1'
            else:
                codes[symbol] +=  '1'
    return codes
```

```
def calculate_codes_df(tree,distr): # para estar en el mismo orden que el dataframe
    states= np.array(distr['states'])
    probab=np.array(distr['probab'])
    codes=dict.fromkeys(states,'')
    for node in reversed(tree):##tre[::-1]
        for symbol in list(node.keys())[0]:
            codes[symbol] +=  '0'
        for symbol in list(node.keys())[1]:
            codes[symbol] +=  '1'

    return pd.DataFrame({'states': states, 'probab': probab,
'code':list(codes.values()), })

def L(distr_coded): # contamos con la normalización anterior de los pesos para da
lugar a frecuencias relativas
    sum = 0
    #print(np.sum(np.array(list(distr_coded['probab']))))
    for i,row in distr_coded.iterrows():
        sum += row['probab']*len(row['code'])
    return sum


def H(distr_coded):
    sum = 0
    for i,row in distr_coded.iterrows():
        sum+=row['probab']*math.log(row['probab'],2)
    return -sum

def check_shannon(distr_coded):
    h=H(distr_coded)
    l=L(distr_coded)
    return h <= l <= h+1

def code(distr_coded,word):
    word_coded=''
    codes=dict(zip(distr_coded['states'], distr_coded['code']))
    for symbol in word:
        word_coded+=codes[symbol]
    return word_coded

dc_en=calculate_codes_df(tree, distr)
print('L',L(dc_en))
print('H',H(dc_en))
print('Shannon(H<L<H+1)',check_shannon(dc_en))

distrib = distr_es
arbol = huffman_tree(distrib)
dc_es=calculate_codes_df(arbol, distrib)
```

```
print('L',L(dc_es))
print('H',H(dc_es))
print('Shannon (H<L<H+1)',check_shannon(dc_es))

code_en=code(dc_en,'medieval')
code_es=code(dc_es,'medieval')
print('medieval in inglés:',code_en)
print('medieval en Spagnolo:',code_es)

dc_es_ord=dc_es.sort_values(by='code', ascending=True)
'''
def check_better_usual(distr_coded):
    sum = 0
    for i,row in distr_coded.iterrows():
        sum+=len(row['code'])
    N=len(distr_coded.index)
    print('suma c_i',sum)
    return sum <= N*math.log(N,2)
def check_better_usual(distr_coded,code,word):
    N=len(distr_coded.index)
    return len(code) <= len(word)*math.log(N,2)
'''
def check_better_usual(distr_coded,code):
    N=len(distr_coded.index)
    return len(code) <= N*math.log(N,2)

print('mejor usual eng',check_better_usual(dc_en,code_en))
print('mejor usual esp',check_better_usual(dc_es,code_es))

def decode(distr_coded,word):
    word_decoded=''
    codes=dict(zip(distr_coded['code'], distr_coded['states']))
    begin=0
    for end in range(len(word)+1):
        symbol=codes.get(word[begin:end])
        if symbol!=None:
            word_decoded+=symbol
            begin=end
    return word_decoded
#Operacion inversa a la codificacion de medieval
#print('decode to
inglés',decode(dc_en,'11110101111110110111111000111011110100110101101110'))
#print('decodificar a
Spagnolo',decode(dc_es,'11000101000010110010100111010100110101'))

print('decode',10111101101110110111011111,'to
inglés:',decode(dc_en,'10111101101110110111011111'))
```