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) , pues los están normalizados en frecuencias, y H(entropía) . También la *comprobación del primer teorema de Shannon .*

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**

1. 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.

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

S\_eng= 11110101111110110111111000111011110100110101101110

S\_esp= 11000101000010110010100111010100110101

Además comprobamos que es eficiente comparando con la codificación binaria usual con .

1. 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 en11101101110110111011111.

**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'))