1. Introducción

En esta práctica hallaremos el código de *Huffman* binario de las variables aleatorias $S_{English}$ y $S_{Español}$ con la distribución de probabilidad respecto a los textos almacenados en "auxiliar_en_pract2.txt" y "auxiliar_es_pract2.txt" respectivamente. También utilizaremos este código para codificar y decodificar palabras.

2. Material empleado

Para esta práctica teníamos un código previo que dado un texto extraía las probabilidades de aparición de cada carácter y generaba el árbol del algoritmo de *Huffman* (como un array de diccionarios de dos elementos que representan a los hijos de cada nodo).

2.1. Hallar el código binario de las dos variables, sus longitudes medias y comprobar que satisfacen el Primer Teorema de Shannon

Para hallar el código de cada carácter dado el árbol, lo que hacemos en la función *extract_code* es recorrer el array en el que está guardado el árbol desde el final (el nodo raiz) hasta el principio. En cada iteración extraemos las dos claves del nodo y añadimos a cada carácter de cada clave un 0 o un 1 dependiendo de a qué nodo pertenezcan.

Este algoritmo extrae correctamente el código binario de *Huffman* del árbol porque para cada carácter lo que estamos haciendo es ir guardando el camino por el cual se bajaría en el árbol hasta llegar a su nodo hoja correspondiente.

Para calcular la longitud media simplemente aplicamos la definición
$L(C) := \frac{1}{W}\sum_{i=1}^{N} w_i|c_i|$ donde $W = \sum_{i=1}^{N} w_i$ es la suma de las frecuencias de cada carácter (en este caso es 1 porque son frecuencias relativas) y $|c_i|$ es la longitud de la cadena que representa a cada carácter.

El cálculo de la entropía también lo hacemos aplicando la definición
$H := -\sum_{j=1}^{N} P_j \log_2 P_j$ donde P_j es la probabilidad de cada carácter.

2.2. Codificar la palabra "fractal" y comprobar la eficiencia de longitud con el código binario usual

Codificar una palabra es muy sencillo ya que teniendo el código de cada carácter lo único que tenemos que hacer es ir añadiendo el código de cada letra.

Para calcular la longitud con el código binario usual necesitamos saber la longitud de cada carácter y esta es $l := \lceil \log_2 N \rceil$ donde N es el número de caracteres del lenguaje. Después multiplicamos por el número de caracteres de la palabra y ya tendríamos la longitud de la palabra.

2.3. Decodificar la palabra "10101000011111011111100" del inglés

Este apartado es similar al anterior, lo único que tenemos que hacer es dar la vuelta al diccionario anterior y decodificar a partir de este nuevo. Este diccionario se puede crear porque cada clave también era única (dos letras no podían tener la misma codificación). Para decodificar, como no sabemos la longitud de cada letra tenemos que ir recorriendo la palabra hasta que una parte coincida con un carácter en el diccionario. Este método funciona porque la clave de un carácter nunca es subclave de otro porque es equivalente a llegar a un nodo hoja (que no tiene hijos).

3. Resultados

```
APARTADO UNO:

Una pequeña muestra del diccionario con la codificación de Senglish: {'u':
'111111', 'b': '1111101', ',': '1111100', 'l': '11110', 't': '1110', ' ': '110',
'a': '1011'}

La longitud media de Senglish es: 4.342324939634357
La entropía de Senglish es:  4.314512109681417
Vemos que se cumple el Teorema de Shannon ya que 4.314512109681417 <=
4.342324939634357 < 5.314512109681417

Una pequeña muestra del diccionario con la codificación de Sspanish: {'\n':
'111111111', 'x': '1111111101', '9': '11111111001', '/': '1111111100011', '2':
'1111111100010', '0': '1111111100001', '=': '1111111100000'}

La longitud media de Sspanish es: 4.214307970337135
La entropía de Sspanish es:  4.183534871433366
Vemos que se cumple el Teorema de Shannon ya que 4.183534871433366 <=
4.214307970337135 < 5.183534871433366


APARTADO DOS:

El codigo binario de la palabra fractal en lengua inglesa es:
01100010010111000111101011111110 y su longitud es: 31
En binario usual sería de longitud: 42


El codigo binario de la palabra fractal en lengua española es:
01101000110001110111100000010 y su longitud es: 30
En binario usual sería de longitud: 42


APARTADO TRES:

La palabra cuyo código binario es: 1010100001111011111100 en inglés es: henal
```

4. Conclusión

En el primer apartado vemos que se cumple el Primer Teorema de Shannon y el resultado está mucho más cerca de la cota inferior que de la superior.

Al comparar la longitud de la codificación en cualquiera de las lenguas de la palabra “fractal” con la longitud en código binario usual vemos que es mucho mejor. Esto sucede en parte porque el código binario de *Huffman* es más eficiente que el binario usual y porque el texto de donde han salido las distribuciones de las variables $S_{English}$ y $S_{Español}$ contiene varias veces la palabra “fractal” lo que hace que las letras que la forman tenga mayor frecuencia y por tanto menor longitud en su codificación.

En el tercer apartado vemos que la longitud de la palabra es de 22 dígitos y en binario usual sería de 30, si quisieramos coger un ejemplo donde la codificación en binario usual sea mejor tendríamos que coger una palabra formada por los caracteres con menor frecuencia pues la longitud de su codificación es mayor que la usual.

A. Calcular el índice de Gini y la diversidad ${}^2D$ de Hill de $S_{English}$

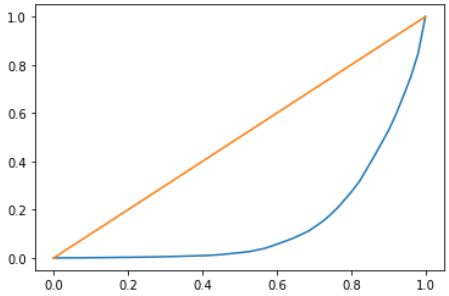
A.1. Material empleado

Para calcular el índice de Gini simplemente tenemos que aplicar la fórmula $GI = 1 - \sum_{j=2}^{N}(y_j + y_{j-1})(x_j - x_{j-1})$ donde x e y son los arrays que definen la curva de Lorenz (esto es que los puntos de la curva son $\{(x_j, y_j)\}_{j=1}^{N}$).

El índice de diversidad de Hill se define como ${}^qD := \lim_{x\to q}(\sum_{j=1}^{N} P_j^x)^{\frac{1}{1-x}}$. Como nosotros queremos calcular la diversidad ${}^2D$ de Hill, el cálculo del límite se reduce a calcular la expresión $(\sum_{j=1}^{N} P_j^2)^{-1}$.

A.2. Resultados

```
APARTADO CUATRO:
```




```
El índice de Gini de Senglish es:  0.7142462918247672
La diversidad 2D de Hill de Senglish es:  14.828886785180586
```

A.3. Conclusión

Hemos dibujado la curva de Lorenz y la gráfica de la recta $y = x$ para hacernos a la idea de como de grande va a ser el valor del índice de Gini cuyo valor es la razón entre el área entre la recta y la curva y el área que hay debajo de la recta. Cúanto mayor sea la distancia entre la curva y la recta, mayor será el índice de Gini lo que indicará que la variable está más lejos de ser equidistribuida.

La diversidad ${}^2D$ de Hill es la inversa del índice de Simpson que en este caso representaría la probabilidad de tomar dos caracteres arbitrarios del texto y que sean el mismo. En el caso particular de $S_{English}$ esta probabilidad sería en torno a un $0,06$.

B. Código

```
"""
Práctica 2
"""

#import os
import numpy as np
import pandas as pd
import math
from itertools import accumulate as acc
import matplotlib.pyplot as plt
from collections import Counter

#### Carpeta donde se encuentran los archivos ####
#ubica = "C:/Users/Python"

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

with open('auxiliar_en_pract2.txt', 'r') as file:
      en = file.read()

with open('auxiliar_es_pract2.txt', 'r') as file:
      es = file.read()

#### Pasamos todas las letras a minúsculas
en = en.lower()
es = es.lower()

#### 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][0] ## Esto proporciona un '0'
list(tree[0].items())[1][0] ## Esto proporciona un '1'

"""
A partir de aquí, código escrito por:
- Jorge Sainero Valle
- Lucas de Torre Barrio
"""

#Extrae el código binario de Huffman de cada carácter y lo devuelve en un
    diccionario {carácter : código}
def extract_code():
    global tree
    d = dict()
    #Empezamos por el final porque es donde está la raíz
    for i in range(tree.size-1,-1,-1):
        elem=tree[i]
        h1=list(elem.keys())[0]
        h2=list(elem.keys())[1]

        for c in h1:
            if c in d:
                d[c]+='0'
            else:
                d[c]='0'

        for c in h2:
            if c in d:
                d[c]+='1'
            else:
                d[c]='1'
    return d

#Calcula la longitud del código binario de Huffman
def longitudMedia(d):
    ac=0
    for i,k in distr.iterrows():
        ac+=len(d[k['states']])*k['probab']
    return ac

#Calcula la entropía de una variable aleatoria con la distribución de
    probabilidades distr['probab']
def entropia():
    h=0
    for p in distr['probab']:
```

```
        h-=p*math.log(p,2)
    return h

def apartado1():
    print("APARTADO UNO:\n")
    global tree
    global distr
    distr = distr_en
    tree = huffman_tree(distr)
    d=extract_code()
    print("Una pequeña muestra del diccionario con la codificación de Senglish:",
        dict(Counter(d).most_common(7)),'\n')
    print ("La longitud media de Senglish es: "+str(longitudMedia(d)))
    h_en=entropia()
    print("La entropía de Senglish es: ",h_en)
    print("Vemos que se cumple el Teorema de Shannon ya que",h_en,"<=",
        str(longitudMedia(d)),"<",h_en+1,"\n")

    distr = distr_es
    tree = huffman_tree(distr)
    d=extract_code()
    print("Una pequeña muestra del diccionario con la codificación de Sspanish:",
        dict(Counter(d).most_common(7)),'\n')
    print ("La longitud media de Sspanish es: "+str(longitudMedia(d)))
    h_es=entropia()
    print("La entropía de Sspanish es: ",h_es)
    print("Vemos que se cumple el Teorema de Shannon ya que",h_es,"<=",
        str(longitudMedia(d)),"<",h_es+1,"\n\n")

#Codifica la palabra pal según el diccionario d
def codificar(pal, d):
    binario=""
    for l in pal:
        binario += d[l]
    return binario

def apartado2():
    print("APARTADO DOS:\n")
    global tree
    global distr
    distr = distr_en
    tree = huffman_tree(distr)
    d_en=extract_code()
    palabra = "fractal"
    pal_bin =codificar(palabra,d_en)
    print("El codigo binario de la palabra fractal en lengua inglesa es:",pal_bin,"
        y su longitud es:",len(pal_bin))
    #La longitud en binario usual es ceil(log2(cantidadDeCracteres)) por cada letra
    print("En binario usual sería de longitud:",
```

```
        len(palabra)*math.ceil(math.log(len(d_en),2)),"\n\n")
    distr = distr_es
    tree = huffman_tree(distr)
    d_es=extract_code()
    pal_bin =codificar(palabra,d_es)
    print("El codigo binario de la palabra fractal en lengua española es:",pal_bin,"
        y su longitud es:",len(pal_bin))
    #La longitud en binario usual es ceil(log2(cantidadDeCracteres)) por cada letra
    print("En binario usual sería de longitud:",
        len(palabra)*math.ceil(math.log(len(d_es),2)),"\n\n")

#Decodifica la palabra pal según el diccionario d
def decodificar(pal,d):
    aux=''
    decode=''
    for i in pal:
        aux+=i
        if aux in d:
            decode+=d[aux]
            aux=''
    return decode

def apartado3():
    print("APARTADO TRES:\n")
    global tree
    global distr
    distr = distr_en
    tree = huffman_tree(distr)
    d_en=extract_code()
    #Invertimos el diccionario para poder decodificar
    d_en_inv=dict(zip(list(d_en.values()),list(d_en.keys())))
    pal_bin='101010000111101111100'
    palabra=decodificar(pal_bin,d_en_inv)
    print("La palabra cuyo código binario es:",pal_bin,"en inglés es:",
        palabra,end='\n\n\n')

#Calcula el índice de Gini de una variable aleatoria con la distribución de
    probabilidades distr['probab']
def gini():
    aux=0
    #ya está ordenado así que no hace falta ordenarlo
    accu=list(acc(distr['probab']))
    plt.plot(np.linspace(0,1,len(accu)),accu)
    plt.plot(np.linspace(0,1,len(accu)),np.linspace(0,1,len(accu)))
    plt.show()
    for i in range(1,len(accu)):
        aux+=(accu[i]+accu[i-1])/len(accu)
    return 1-aux
```

```
#Calcula la diversidad 2D de Hill de una variable aleatoria con la distribución
    de probabilidades distr['probab']
def diver2hill():
    aux=0
    for p in distr['probab']:
        aux+=p*p
    return 1/aux

def apartado4():
    print("APARTADO CUATRO:\n")
    global distr
    distr = distr_en
    print("El índice de Gini de Senglish es: ",gini())
    print("La diversidad 2D de Hill de Senglish es: ",diver2hill())
    distr = distr_es
    print("El índice de Gini de Sspanish es: ",gini())
    print("La diversidad 2D de Hill de Sspanish es: ",diver2hill())

apartado1()
apartado2()
apartado3()
apartado4()
```