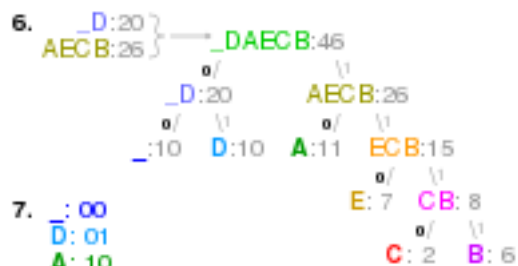
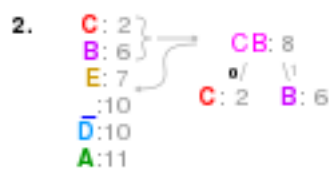


GEOMETRÍA COMPUTACIONAL

CÓDIGO HUFFMAN

1. "A_DEAD_DAD_CEDDED_A_BAD_BABE_A_BEADED_ABACA_BED"



8. "10000111010010001100100111011001110010001111100100111101111100010001111110100111010011101001011111011101000111111001"

LUCAS DE TORRE

Índice

1. Introducción	2
2. Material	2
3. Resultados	2
4. Conclusiones	3
5. Anexo A: Código	4

1. Introducción

Dadas las variables aleatorias $S_{English} = \{a,b,c,...,!,?,0,...,9\}$ y $S_{Español} = \{a,b,c,...,\tilde{n},...,!,?,0,...,9\}$ y su distribución de probabilidad (tomada de los archivos “auxiliar_enpract2.txt” y “auxiliar_es_pract2.txt”, respectivamente), buscamos hallar el código Huffman binario de ambas variables, sus longitudes medias y comprobar que verifican el Primer Teorema de Shannon. Además, usaremos estos códigos binarios para codificar y decodificar palabras. Adicionalmente, calcularemos el índice de Gini y la diversidad 2D de Hill.

2. Material

Partimos de un código que, para cada variable aleatoria, guarda en un array el árbol de Huffman dado por la distribución de probabilidad de dicha variable. A partir de dichos árboles, creamos un diccionario por cada una de las dos variables que, dado un elemento de nuestra variable aleatoria, devuelva la codificación Huffman binaria de dicho elemento. Estos diccionarios los creamos recorriendo los árboles desde la raíz hasta las hojas, hallando así el código Huffman binario de cada elemento.

Después, calculamos la longitud media de nuestras variables $L(S)$. Para ello, simplemente multiplicamos la longitud de cada elemento por su probabilidad y sumamos los resultados.

Ahora, calculamos la entropía de nuestros sistemas: $H(S) = -\sum_{j=1}^N P_j \log_2 P_j$, donde N es el número de elementos del sistema y P_j es la probabilidad asociada al elemento j -ésimo. Podemos verificar que se cumple el Primer Teorema de Shannon, según el cual, $H(S) \leq L(S) < H(S) + 1$.

Procedemos a codificar en ambas lenguas la palabra “fractal”. Únicamente necesitamos hacer uso del diccionario anterior, que nos da, para cada letra de la palabra, su codificación. Una vez calculada la palabra, comparamos la eficiencia de esta codificación en comparación con la codificación binaria usual. Esto lo hacemos comparando el número de bits necesarios en cada caso.

A continuación decodificamos la cadena “000011101111111111010”. Comenzamos creando un diccionario que asocie a cada cadena de bits el carácter correspondiente (es el diccionario que teníamos pero intercambiando las claves por los valores). Después recorreremos los bits a decodificar hasta encontrar una secuencia que coincida con una entrada de nuestro diccionario: por ejemplo, en este caso cogemos el primer 0 y vemos si coincide con una entrada de nuestro diccionario; si no, cogemos el siguiente elemento (0) y comprobamos si 00 tiene entrada en nuestro diccionario, y proseguimos hasta que así sea. Una vez encontrado, este será un carácter de la palabra decodificada, por lo que repetimos el proceso con el resto de la cadena hasta que esta sea vacía o ya no queden más elementos por mirar en la misma y tendremos la palabra buscada.

Proseguimos calculando el índice de Gini de nuestras variables. Utilizamos la fórmula $GI = 1 - \sum_{j=2}^N (y_j + y_{j-1})(x_j + x_{j-1} - 1)$, siendo y_j la frecuencia acumulada de aparición de todas las letras anteriores y la j -ésima, estando las letras ordenadas de menor a mayor probabilidad de aparición. Por último, calculamos la diversidad 2D de Hill mediante la fórmula $^2D = \sum_{j=1}^N P_j^2$.

3. Resultados

Una vez hallado el diccionario, obtenemos fácilmente el código binario de cada letra. Por ejemplo, para $S_{English}$, tenemos que $a = 1011$ o que $u = 111111$. Las longitudes medias son:

- $L(S_{English}) = 4,342324939634357$ (que verifica el Primer Teorema de Shannon porque la entropía es $H(S_{English}) = 4,314512109681417 \leq 4,342324939634357 < H(S_{English}) + 1$)
- $L(S_{Español}) = 4,214307970337135$ (que verifica el Primer Teorema de Shannon porque la entropía es $H(S_{Español}) = 4,183534871433366 \leq 4,214307970337135 < H(S_{Español}) + 1$).

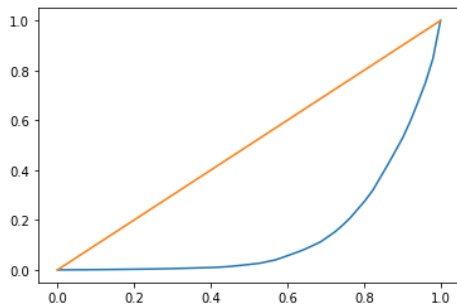
Al codificar la palabra “fractal” con el diccionario creado a partir del árbol de Huffman obtenemos el

código:

- $S_{English}$: 011000100101110001111010111110, que tiene 31 bits.
- $S_{Español}$: 011010001100011101111100000010, que tiene 30 bits.

En ambos casos vemos que este código es más eficiente que el binario usual, en el cual esta palabra necesita 42 bits.

En el cálculo del índice de Gini y a la diversidad 2D de Hill para las dos variables se obtiene:



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

- $S_{English}$: El índice de Gini es $GI = 0,7142462918247$ y la diversidad 2D de Hill es $^2D = 14,828886785180$.
- $S_{Español}$: El índice de Gini es $GI = 0,7332199409333$ y la diversidad 2D de Hill es $^2D = 13,620973624876$.

Se observa que los valores obtenidos para ambas variables son similares.

4. Conclusiones

5. Anexo A: Código

```

1  """
2  Pr ctica 2
3  """
4
5  #import os
6  import numpy as np
7  import pandas as pd
8  import math
9  from itertools import accumulate as acc
10 import matplotlib.pyplot as plt
11 from collections import Counter
12
13 ##### Carpeta donde se encuentran los archivos #####
14 #ubica = "C:/Users/Python"
15
16 ##### Vamos al directorio de trabajo#####
17 #os.getcwd()
18 #os.chdir(ubica)
19 #files = os.listdir(ubica)
20
21 with open('auxiliar_en_pract2.txt', 'r') as file:
22     en = file.read()
23
24 with open('auxiliar_es_pract2.txt', 'r') as file:
25     es = file.read()
26
27 ##### Pasamos todas las letras a min sculas
28 en = en.lower()
29 es = es.lower()
30
31 ##### Contamos cuantas letras hay en cada texto
32 from collections import Counter
33 tab_en = Counter(en)
34 tab_es = Counter(es)
35
36 ##### Transformamos en formato array de los car cteres (states) y su frecuencia
37 ##### Finalmente realizamos un DataFrame con Pandas y ordenamos con 'sort'
38 tab_en_states = np.array(list(tab_en))
39 tab_en_weights = np.array(list(tab_en.values()))
40 tab_en_probab = tab_en_weights/float(np.sum(tab_en_weights))
41 distr_en = pd.DataFrame({'states': tab_en_states, 'probab': tab_en_probab})
42 distr_en = distr_en.sort_values(by='probab', ascending=True)
43 distr_en.index=np.arange(0,len(tab_en_states))
44
45 tab_es_states = np.array(list(tab_es))
46 tab_es_weights = np.array(list(tab_es.values()))
47 tab_es_probab = tab_es_weights/float(np.sum(tab_es_weights))
48 distr_es = pd.DataFrame({'states': tab_es_states, 'probab': tab_es_probab })
49 distr_es = distr_es.sort_values(by='probab', ascending=True)
50 distr_es.index=np.arange(0,len(tab_es_states))
51
52 ##### Para obtener una rama, fusionamos los dos states con menor frecuencia
53 distr = distr_en
54 ' '.join(distr['states'][[0,1]])
55
56 ### Es decir:
57 states = np.array(distr['states'])
58 probab = np.array(distr['probab'])
59 state_new = np.array([' '.join(states[[0,1]])]) #Ojo con: state_new.ndim
60 probab_new = np.array([np.sum(probab[[0,1]])]) #Ojo con: probab_new.ndim

```

```

61 codigo = np.array([states[0]: 0, states[1]: 1])
62 states = np.concatenate((states[np.arange(2,len(states))], state_new), axis=0)
63 probab = np.concatenate((probab[np.arange(2,len(probab))], probab_new), axis=0)
64 distr = pd.DataFrame({'states': states, 'probab': probab, })
65 distr = distr.sort_values(by='probab', ascending=True)
66 distr.index=np.arange(0,len(states))
67
68 #Creamos un diccionario
69 branch = {'distr':distr, 'codigo':codigo}
70
71 ## Ahora definimos una funci n que haga ex ctamente lo mismo
72 def huffman_branch(distr):
73     states = np.array(distr['states'])
74     probab = np.array(distr['probab'])
75     state_new = np.array([''.join(states[[0,1]])])
76     probab_new = np.array([np.sum(probab[[0,1]])])
77     codigo = np.array([states[0]: 0, states[1]: 1])
78     states = np.concatenate((states[np.arange(2,len(states))], state_new), axis=0)
79     probab = np.concatenate((probab[np.arange(2,len(probab))], probab_new), axis=0)
80     distr = pd.DataFrame({'states': states, 'probab': probab, })
81     distr = distr.sort_values(by='probab', ascending=True)
82     distr.index=np.arange(0,len(states))
83     branch = {'distr':distr, 'codigo':codigo}
84     return(branch)
85
86 def huffman_tree(distr):
87     tree = np.array([])
88     while len(distr) > 1:
89         branch = huffman_branch(distr)
90         distr = branch['distr']
91         code = np.array([branch['codigo']])
92         tree = np.concatenate((tree, code), axis=None)
93     return(tree)
94
95 distr = distr_en
96 tree = huffman_tree(distr)
97 tree[0].items()
98 tree[0].values()
99
100 #Buscar cada estado dentro de cada uno de los dos items
101 list(tree[0].items())[0][0] ## Esto proporciona un '0'
102 list(tree[0].items())[1][0] ## Esto proporciona un '1'
103
104 """
105 A partir de aqu , c odigo escrito por:
106 - Jorge Sainero Valle
107 - Lucas de Torre Barrio
108 """
109
110 #Extrae el c digo binario de Huffman de cada car cter y lo devuelve en un
    diccionario {car cter : c digo}
111 def extract_code():
112     global tree
113     d = dict()
114     #Empezamos por el final porque es donde est la ra z
115     for i in range(tree.size-1,-1,-1):
116         elem=tree[i]
117         h1=list(elem.keys())[0]
118         h2=list(elem.keys())[1]
119
120         for c in h1:
121             if c in d:

```

```

122         d[c] += '0'
123     else:
124         d[c] += '1'
125
126     for c in h2:
127         if c in d:
128             d[c] += '1'
129         else:
130             d[c] = '1'
131     return d
132
133 #Calcula la longitud del c digo binario de Huffman
134 def longitudMedia(d):
135     ac=0
136     for i,k in distr.iterrows():
137         ac+=len(d[k['states']])*k['probab']
138     return ac
139
140 #Calcula la entropía de una variable aleatoria con la distribución de
141 #probabilidades distr['probab']
142 def entropia():
143     h=0
144     for p in distr['probab']:
145         h-=p*math.log(p,2)
146     return h
147
148
149 def apartado1():
150     print("APARTADO UNO:\n")
151     global tree
152     global distr
153     distr = distr_en
154     tree = huffman_tree(distr)
155     d=extract_code()
156     print("Una pequeña muestra del diccionario con la codificación de Senglish:",
157           dict(Counter(d).most_common(7)),'\n')
158     print("La longitud media de Senglish es: "+str(longitudMedia(d)))
159     h_en=entropia()
160     print("La entropía de Senglish es: ",h_en)
161     print("Vemos que se cumple el Teorema de Shannon ya que",h_en,"<=",str(
162           longitudMedia(d)),"< ",h_en+1,"\n")
163
164     distr = distr_es
165     tree = huffman_tree(distr)
166     d=extract_code()
167     print("Una pequeña muestra del diccionario con la codificación de Sspanish:",
168           dict(Counter(d).most_common(7)),'\n')
169     print("La longitud media de Sspanish es: "+str(longitudMedia(d)))
170     h_es=entropia()
171     print("La entropía de Sspanish es: ",h_es)
172     print("Vemos que se cumple el Teorema de Shannon ya que",h_es,"<=",str(
173           longitudMedia(d)),"< ",h_es+1,"\n\n")
174
175 #Codifica la palabra pal según el diccionario d
176 def codificar(pal, d):
177     binario=""
178     for l in pal:
179         binario += d[l]
180     return binario

```

```

179 def apartado2():
180     print("APARTADO DOS:\n")
181     global tree
182     global distr
183     distr = distr_en
184     tree = huffman_tree(distr)
185     d_en=extract_code()
186     palabra = "fractal"
187     pal_bin =codificar(palabra,d_en)
188     print("El codigo binario de la palabra fractal en lengua inglesa es:",pal_bin,"y
su longitud es:",len(pal_bin))
189     #La longitud en binario usual es ceil(log2(cantidadDeCracteres)) por cada letra
190     print("En binario usual ser a de longitud:", len(palabra)*math.ceil(math.log(len
(d_en),2)), "\n\n")
191     distr = distr_es
192     tree = huffman_tree(distr)
193     d_es=extract_code()
194     pal_bin =codificar(palabra,d_es)
195     print("El codigo binario de la palabra fractal en lengua espa ola es:",pal_bin,"
y su longitud es:",len(pal_bin))
196     #La longitud en binario usual es ceil(log2(cantidadDeCracteres)) por cada letra
197     print("En binario usual ser a de longitud:", len(palabra)*math.ceil(math.log(len
(d_es),2)), "\n\n")
198
199 #Decodifica la palabra pal seg n el diccionario d
200 def decodificar(pal,d):
201     aux=' '
202     decode=' '
203     for i in pal:
204         aux+=i
205         if aux in d:
206             decode+=d[aux]
207             aux=' '
208     return decode
209
210 def apartado3():
211     print("APARTADO TRES:\n")
212     global tree
213     global distr
214     distr = distr_en
215     tree = huffman_tree(distr)
216     d_en=extract_code()
217     #Invertimos el diccionario para poder decodificar
218     d_en_inv=dict(zip(list(d_en.values()),list(d_en.keys()))))
219     pal_bin='1010100001111011111100'
220     palabra=decodificar(pal_bin,d_en_inv)
221     print("La palabra cuyo c digo binario es:",pal_bin,"en ingl s es:",palabra,end=
'\n\n\n')
222
223 #Calcula el ndice de Gini de una variable aleatoria con la distribuci n de
probabilidades distr['probab']
224 def gini():
225     aux=0
226     #ya est ordenado as que no hace falta ordenarlo
227     accu=list(acc(distr['probab']))
228     plt.plot(np.linspace(0,1,len(accu)),accu)
229     plt.plot(np.linspace(0,1,len(accu)),np.linspace(0,1,len(accu)))
230     plt.show()
231     for i in range(1,len(accu)):
232         aux+=(accu[i]+accu[i-1])/len(accu)
233     return 1-aux
234

```



```
235 #Calcula la diversidad 2D de Hill de una variable aleatoria con la distribuci n de
    probabilidades distr['probab']
236 def diver2hill():
237     aux=0
238     for p in distr['probab']:
239         aux+=p*p
240     return 1/aux
241
242
243 def apartado4():
244     print("APARTADO CUATRO:\n")
245     global distr
246     distr = distr_en
247     print("El ndice de Gini de Senglish es: ",gini())
248     print("La diversidad 2D de Hill de Senglish es: ",diver2hill())
249     distr = distr_es
250     print("El ndice de Gini de Sspanish es: ",gini())
251     print("La diversidad 2D de Hill de Sspanish es: ",diver2hill())
252
253 apartado1()
254 apartado2()
255 apartado3()
256 apartado4()
```