Práctica 1 GCOMP: Código Huffman y 1er Teorema de Shannon

Juan Carlos Díaz Rodríguez

1. Introducción (motivación/objetivo de la práctica)

Durante el inicio del curso hemos visto el concepto de entropía de un sistema. Este concepto está fuertemente ligado con el número de bits que nos haría falta para representar el sistema. En esta práctica pretendemos ver un algoritmo para obtener una codificación óptima de un texto dado, representando por tanto un sistema con una codificación binaria. En ella corroboraremos que los árboles de *Huffman* dan una longitud dentro de la optimalidad propuesta por el *Teorema de Shannon* y analizaremos la eficiencia de compresión con respecto a la codificación usual en ASCII.

2. Material usado (método y datos)

El algoritmo seguido es aquel propuesto en las notas de la asignatura, donde adaptamos el pseudo-código con las siguientes estructuras de datos:

- **Árbol binario**. La construcción del árbol de *Huffman* se realiza con una estructura de árbol binario. Al no necesitar un comportamiento muy elaborado, basta con ir creando 4-uplas con la información necesaria en cada nodo. Por motivos que se explicarán más adelante el primer elemento debe ser aquel que marque el orden de los datos. Por lo tanto se trata de un par `(#apariciones, orden de aparición)`, seguido de los caracteres que representa dicho nodo, y dos punteros a los hijos izquierdo y derecho respectivamente.
- **Cola de prioridad**. Esta funcionalidad se obtiene con la librería `heapq`. Esta estructura de datos nos permite ir obteniendo los nodos con menor número de apariciones acumuladas en un tiempo de $\mathcal{O}(\log(n))$. Esta librería representa una cola de prioridad mediante una lista. Todas sus operaciones hacen uso del orden por defecto que esté definido para el tipo de datos que contiene. Por lo tanto, es importante que todos los elementos que introduzcamos en esta lista sean del mismo tipo y comparables. Dado que nuestros nodos son tuplas, el orden por defecto de estas se realiza comparando los elementos de izquierda a derecha. Como hemos comentado anteriormente, hemos elegido el primero elemento para que tenga un orden estricto, por lo que basta la comparación de este primer par para obtener el orden de los datos (y por ello era importante que el orden estuviese en el primer elemento de la 4-upla).
- **Diccionario**. Una vez obtenido el árbol de *Huffman*, crearemos un diccionario recorriéndolo recursivamente. Indexaremos mediante los caracteres del sistema los códigos que le correspondan según el algoritmo empleado anteriormente. Suponiendo una implementación de los diccionarios de *Python* mediante tablas Hash, el tiempo de acceso al código de cada caracter se realiza en tiempo constante.

Al contrario que en la plantilla proporcionada, dado un empate en el número de apariciones, determinaremos si un nodo es menor que otro teniendo en cuenta el orden de aparición de los caracteres. Cuando unamos dos nodos, tomaremos como orden de aparición el mínimo de los órdenes de sus hijos. De este modo el orden generado es estricto y completamente reproducible. Para poder hacer una codificación generaremos un diccionario indexado por el caracter buscado que nos proporcione el código correspondiente.

Las estructuras de codificación han sido calculadas a partir de los textos de prueba dados.

3. Resultados

Comentamos a continuación los resultados de los distintos apartados pedidos.

Apartado i

Tras obtener el árbol con la función `build_tree(heap)` y el diccionario de traducción con `build_dictionary(tree)`, podemos obtener las codificaciones de ambos textos. No se pone en esta memoria el código calculado por razones de espacio. Las longitudes medias obtenidas son:

$$L\left(S_{Eng}\right) = 4{,}437 \quad L\left(S_{Esp}\right) = 4{,}371\ ,$$

mientras que las entropías de los sistemas son:

$$H\left(S_{Eng}\right) = 4{,}412 \quad H\left(S_{Esp}\right) = 4{,}338\ .$$

Se comprueba por tanto el 1er Teorema de Shannon, además de la alta optimalidad de los códigos de *Huffman*.

Apartado ii

En este apartado trabajamos con la palabra *dimension* para ambos lenguajes. Obtenemos las codificaciones dadas por cada diccionario:

Codificación en inglés : 00111011101100000100001000111010110000

Codificación en español: 11110011011101100010011010011010001001,

de longitudes 37 y 38 respectivamente. Si tenemos en cuenta la codificación ASCII la longitud de estos códigos habría sido de 54 dígitos binarios, por lo que esto supone una mejora del 45,95 % y del 42,11 % respectivamente.

Apartado iii

Queremos ahora obtener la palabra resultado de decodificar la cadena 010101000110011100110111100010111111101010100011110. Dado que esta codificación habrá sido obtenida con un árbol distinto al aquí presentado, el resultado no es el esperado.

En este caso se obtiene la palabra 'osml rphosi'. Se entiende que la palabra buscada era 'isomorphism', para la cual el código correcto para este árbol es: 011101000101011000101111100010111110111010001100.

4. Conclusión

El 1er Teorema de Shannon nos señala que no existe la compresión perfecta. Uno no puede expresar un sistema con una codificación tan corta como se proponga pues este resultado nos da una cota inferior para ello. Sin embargo, existen algoritmos relativamente sencillos que nos permiten acercarnos mucho a la cota dada. Los códigos de *Huffman* nos dan una solución a este problema. Es de destacar, que la compresión obtenida es variable según la palabra codificada. Si la palabra que elegimos no sigue la distribución de muestra que se le dio al algoritmo para crear el traductor, la compresión no será tan óptima. Los resultados obtenidos nos permiten ver cómo con palabras comunes rondamos mejoras del 40 %.

Hemos podido comprobar también la importancia de establecer un orden controlado. Desconocer el método de ordenación que emplea un algoritmo, como es el orden que establece la librería `Pandas` para ordenar, hace que no seamos capaces de reproducir el árbol dado por otros programas. Esto, por tanto, hace que las traducciones no sean coherentes.

A. Código

Se presenta aquí el código creado para la realización de la práctica. Las primeras funciones son aquellas empleadas para crear los árboles y diccionarios mencionados en la memoria, mientras que el resto del script permite que mediante una ejecución completa se puedan ver en pantalla los resultados comentados anteriormente.

```
import os
import numpy as np
import heapq as hq

def build_tree(heap):
    # Modelo del arbol:
    # 4-uplas donde cada posicione es:
    #   0: (n mero de apariciones, orden de aparicion)
    #   1: nombre del nodo
    #   2: hijo izquierdo
    #   3: hijo derecho
    while len(heap) > 1:
        item1 = hq.heappop(heap)
        item2 = hq.heappop(heap)
        result = (\
            (item1[0][0] + item2[0][0], min(item1[0][1], item2[0][1])),\
            item1[1] + item2[1],\
            item1,\
            item2)
        hq.heappush(heap, result)
    # El  ltimo  elemento en la cola de prioridad tendr  el
    #  rbol  completo construido
    return hq.heappop(heap)

def build_dictionary(tree):
    if tree[2] == None: # Entonces tree[3] tambi n ser  None
        return {tree[1]: ''}
    left = build_dictionary(tree[2])
    for k in left.keys():
        left[k] = '0' + left[k]
    right = build_dictionary(tree[3])
    for k in right.keys():
        right[k] = '1' + right[k]
    return {**left, **right}

def code_from_text(text, dict):
    code = ''
    for c in text:
        if not c in dict:
            code += '('+c+' no esta en el diccionario)'
        else:
            code += dict[c]
    return code

def text_from_code(code, tree):
    i = 0
    text = ''
    while i < len(code):
        curr_node = tree
        not_found = True
        while not_found:
            try:
                if code[i] == '0':
                    curr_node = curr_node[2]
                else:
                    curr_node = curr_node[3]
                i += 1
                if curr_node[2] == None:
                    not_found = False
            except IndexError:
                # Si el c digo no es consistente con el  rbol  dado
                # saltar  un  ndice  de error. Devolveremos la palabra
                # decodificada hasta el momento
                return text
        text += curr_node[1]
    return text


os.getcwd()
```

```
#os.chdir(ruta)
#files = os.listdir(ruta)

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()


from collections import Counter
tab_en = Counter(en)
tab_es = Counter(es)
# A adimos el orden de aparicion
for k in tab_en.keys():
    tab_en[k] = [tab_en[k], -1]
for k in tab_es.keys():
    tab_es[k] = [tab_es[k], -1]
for i in range(len(en)):
    if tab_en[en[i]][1] == -1:
        tab_en[en[i]][1] = i
        tab_en[en[i]] = tuple(tab_en[en[i]])
for i in range(len(es)):
    if tab_es[es[i]][1] == -1:
        tab_es[es[i]][1] = i
        tab_es[es[i]] = tuple(tab_es[es[i]])
# Apartado 1
print('Apartado 1')
# Hallar el c digo de Huffman binario y las logitudes medias de ambos textos.
# Comprobar que se satisface el Teorema de Shannon.

# Primero construimos los   rboles   binarios
list_en = [(v, k, None, None) for k, v, in tab_en.items()]
list_es = [(v, k, None, None) for k, v, in tab_es.items()]

hq.heapify(list_en)
hq.heapify(list_es)

tree_en = build_tree(list_en)
tree_es = build_tree(list_es)

dict_en = build_dictionary(tree_en)
dict_es = build_dictionary(tree_es)

code_en = code_from_text(en, dict_en)
code_es = code_from_text(es, dict_es)


# Calculamos las longitudes medias:
W_en = np.sum([v[0] for v in tab_en.values()])
W_es = np.sum([v[0] for v in tab_es.values()])

L_en = 1/float(W_en) * (np.sum([v[0] * len(dict_en[k]) for k, v, in tab_en.items()
    ]))
L_es = 1/float(W_es) * (np.sum([v[0] * len(dict_es[k]) for k, v, in tab_es.items()
    ]))
print('Calculo de las longitudes medias:')
print('L(Seng) = ' + str(L_en))
print('L(Sesp) = ' + str(L_es))
print('')

# Calculo de la entropia

prob_en = {k: float(v[0]) / W_en for k, v, in tab_en.items()}
prob_es = {k: float(v[0]) / W_es for k, v, in tab_es.items()}

H_en = -np.sum([v * np.log2(v) for v in prob_en.values()])
H_es = -np.sum([v * np.log2(v) for v in prob_es.values()])

print('C lculo de la entropia')
print('H(Seng) = ' + str(H_en))
print('H(Sesp) = ' + str(H_es))
print('')

if L_en >= H_en and L_en < H_en + 1 and L_es >= H_es and L_es < H_es + 1:
    print('Se cumple el Teorema de Shannon')
```

```
144 else:
145     print('NO se cumple el Teorema de Shannon')
146 print('')
147 # Apartado 2
148 # Codificar la palabra 'dimension' en ambas lenguas
149 
150 word = 'dimension'
151 code_en = code_from_text(word, dict_en)
152 code_es = code_from_text(word, dict_es)
153 print('Apartado 2')
154 print('Codificaci n en ingl s : ' + code_en)
155 print('Codificaci n en espa ol: ' + code_es + '\n')
156 long_en = int(len(word) * np.ceil(np.log2(len(tab_en))))
157 long_es = int(len(word) * np.ceil(np.log2(len(tab_es))))
158 print('Longitud del c digo en ingl s: ' + str(len(code_en)))
159 print('Longitud de la codificaci n usual: ' + str(long_en))
160 print('Mejora: ' + f'{((long_en / len(code_en) - 1)*100):.2f}' + '%\n')
161 print('Longitud del c digo en espa ol: ' + str(len(code_es)))
162 print('Longitud de la codificaci n usual: ' + str(long_es))
163 print('Mejora: ' + f'{((long_es / len(code_es) - 1)*100):.2f}' + '%\n')
164 # Apartado 3
165 print('Apartado 3')
166 code_3 = '0101010001100111001101111000101111110101010001110'
167 print('Resultado de la decodificaci n: ' + text_from_code(code_3, tree_en))
```

Listing 1: Código de la prática 1