

## Práctica 1

### Teoría de la Información. Codificación de fuente

#### Resumen

Comenzaremos esta práctica repasando el concepto de entropía, para a continuación ver dos ejemplos de codificadores de fuente, como son los códigos Huffman y los códigos LZW (Lempel-Ziv-Welch).

## 1 Introducción

En primer lugar debemos desarrollar las siguientes funciones en Matlab:

1. Generar una función que, dado un vector de probabilidades  $p$  de una determinada fuente de información, devuelva el valor de la entropía para esa fuente.

```
function H = entropia(p)
```

2. Generar una función que, a partir de un vector de probabilidades  $p$  de una determinada fuente de información, calcule el código Huffman correspondiente.

```
function codigo = huffman(p)
```

Como alternativa, se puede utilizar la función `huffmandict` de Matlab. Se necesitan dos parámetros: los símbolos a transmitir, que en nuestro caso pueden ser directamente `1:6`, y las probabilidades de cada símbolo. Podéis comprobar el funcionamiento de las funciones creadas considerando una fuente con 6 símbolos posibles con probabilidades:  $p = [0,1, 0,3, 0,05, 0,09, 0,21, 0,25]$  La entropía de esta fuente es de 2,3549 bits por símbolo de fuente, la longitud media de las palabras del código Huffman es de 2,38 bits, y la eficiencia 0,9895.

3. Generar una función que codifique un determinado mensaje utilizando el algoritmo LZW utilizando un diccionario inicial dado.

```
function codigo = lzw(cadena, diccionario)
```

4. Generar la función que sea capaz de decodificar un mensaje codificado con LZW tomando como dato el diccionario inicial.

```
function mensaje = lzwdec(codigo, diccionario)
```

Para probar el algoritmo de codificación LZW, podéis utilizar un diccionario que conste únicamente de dos símbolos:  $\text{diccionario} = 'A', 'B'$ , y codificar la palabra `ABAABABA`. El resultado deberían de ser 6 códigos:

```
codigo = [1 2 1 3 6 ]
```

Vamos a considerar a partir de ahora un diccionario consistente en todas las letras del español en minúscula, con sus correspondientes probabilidades de ocurrencia, que se suministran en las variables `diccionario` y  $p$  del archivo `Datos.mat`:

Carácter	Probabilidad
ESPACIO	0.1899
a	0.0934
b	0.0179
c	0.0326
d	0.0406
e	0.0987
f	0.0056
g	0.0143
h	0.0057
i	0.0506
j	0.0040
k	0.0001
l	0.0402
m	0.0256
n	0.0544
o	0.0703
p	0.0203
q	0.0071
r	0.0557
s	0.0646
t	0.0375
u	0.0237
v	0.0092
w	0.0001
x	0.0017
y	0.0082
z	0.0038
á	0.0041
é	0.0035
í	0.0059
ó	0.0067
ú	0.0014
ü	0.0001
ñ	0.0025

5. Una vez que tenemos listo todo lo anterior, vamos a realizar las siguientes pruebas:

- Considerar una fuente con probabilidades de símbolo:

$$p = \left[ \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}, \frac{1}{128}, \frac{1}{256}, \frac{1}{256} \right]$$

Calcular la eficiencia de este código. ¿Qué debe ocurrir para que un código tenga eficiencia 1?

- Calcular la entropía del alfabeto español con los datos dados.
- Obtener el código Huffman correspondiente a las letras del español, calculando la longitud media de palabra, la tasa de compresión y la eficiencia.
- ¿Qué sucedería si suponemos ahora que las letras se van a agrupar de dos en dos?. Supondremos que la fuente de información no tiene memoria, de forma que la probabilidad de aparición de un par de letras es igual al producto de las probabilidades individuales.
- Codificar, utilizando el algoritmo LZW, la cadena de texto mensaje incluida en el archivo *Datos.mat*. Obtener, como antes, la longitud media de palabra, la tasa de compresión y la eficiencia obtenidas.
- Decodificar el mensaje decodificado, comprobando que se obtiene de nuevo el mensaje original.

## 2 Pistas

- Si os animáis a intentar programar el codificador Huffman, es muy cómodo trabajar con números enteros, y en el último momento traducirlos a binario con la función `dec2bin`. Añadir un 0 a la derecha de un número binario no es más que multiplicarlo por dos en decimal, y añadir un 1, multiplicar por 2 y sumar 1.
- El diccionario del codificador LZW os lo paso en formato de *cell array* porque creo que es lo más cómodo, ya que cada entrada del diccionario puede tener una longitud distinta.
- El comando `strcmp(cadena,diccionario)` resulta muy útil para localizar si una determinada cadena existe en el diccionario, y dónde se encuentra.
- Para el caso de la codificación del texto por pares de caracteres es útil tener en cuenta la función `kron`.

## 3 ¿Qué entregar?

- Código de las funciones generadas