Práctica 1
Teoría de la Información. Codificación de fuente

Resumen de la práctica

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 el directorio de Python hay varios ficheros que utilizaremos para esta práctica:

- `Practica1a.py`, `Practica1b.py` y `Practica1c.py`: Se trata de los archivos principales que debemos ejecutar para desarrollar la práctica. Se refieren, respectivamente, a ejemplos de codificación fuente con un archivo de texto, uno de audio y una imagen.
- `CodFuente.py`. Archivo con una serie de funciones relacionadas con la codificación fuente:
  - **huffman**
  - **huffman_cod**
  - **huffman_dec**
  - **lzw_cod**
  - **lzw_dec**

Dentro del archivo *FuncionesP1.py* veréis que faltan por desarrollar algunas funciones que se han dejado en blanco:

- **gen_huffman_dict**: Esta función debe tomar a su entrada un mensaje en formato *string*, *List* ó *ndarray* (array de NumPy), en función del tipo de archivo con el que estemos trabajando, y devolver a su salida un diccionario con pares (símbolo, probabilidad). Algunas sugerencias para esta función:
  - `np.ravel(a)`: Esta función convierte el array *a* de más de una dimensión (e.g. una imagen) en un array unidimensional.
  - `list(x)`: Crea una lista a partir de la variable que le demos. Por ejemplo, si es una cadena, genera una lista con cada carácter de la cadena por separado.
  - `collections.Counter(x)`: A partir de una lista, devuelve el recuento del número de ocurrencias de cada elemento de la lista.
  - `dict(x)`: Crea un diccionario a partir de un contador como el del punto anterior.
- **gen_lzw_dict**: Función muy similar a la anterior, pero esta vez la salida es un diccionario con pares (símbolo, código), donde el código es simplemente un número entero que enumera cada uno de los posibles elementos de la entrada.
- **entropia**: Esta función debe tomar a su entrada un mensaje en formato *string* ó *List*, y devolver a su salida el valor de la entropía para dicha fuente. Podéis aprovecharos de la función *gen_huffman_dict* para coger de ahí las probabilidades de cada uno de los símbolos.
- **tamanoOriginal**: Devuelve el tamaño, en KB, de un mensaje de entrada al sistema. Toma como entradas el propio mensaje y el número de bits por cada símbolo del mensaje, que serán 8 para el caso de iamgen o texto y 16 para audio.
- **tamanoMinimo**: Devuelve el tamaño mínimo, en KB, que se podría conseguir si alcanzásemos una eficiencia de codificación máxima.

- **tamanoCodificado**: Devuelve el tamaño, en KB, de una secuencia de bits dada.

Se puede comprobar el funcionamiento de estas funciones utilizando la cadena de datos "una prueba". Se debería obtener una entropía de 2.92 y los siguientes diccionarios:

- Huffman:

```
{'u': 0.2, 'n': 0.1, 'a': 0.2, ' ': 0.1, 'p': 0.1, 'r': 0.1, 'e': 0.1, 'b': 0.1}
```

- LZW:

```
{'b': 7, 'e': 6, 'r': 5, 'p': 4, ' ': 3, 'a': 2, 'n': 1, 'u': 0}
```

2 Archivo de texto

Una vez hayáis completado estas funciones ya podéis abrir el fichero `Practica1a.py` y ejecutarlo. Comprobad que el fichero de salida (salida.txt) coincida con la entrada, y los valores que obtenéis de los tamaños de los archivos en cada fase tanto para el caso de utilizar codificación Huffman como codificación LZW.

3 Archivo de audio

Para el caso de archivos de audio (y de imagen, como veremos más adelante), es posible utilizar algunas estrategias ligeramente distintas a las que hemos visto. En estos casos ese habla de codificación con pérdidas (lo que hacen los sistemas mp3, aac, mp4 y similares), ya que vamos a reducir el tamaño del archivo, al igual que hemos hecho antes, pero en este caso vamos a asumir que el archivo resultante no sea igual que el original, aunque sí lo suficientemente parecido como para que las diferencias no sean perceptibles. Este tipo de codificadores se denominan a veces "codificadores de destino" ya que basan gran parte de su funcionamiento en las características del receptor (en el caso del audio, el oído humano).

La idea es muy simple. Previamente a la codificación Huffman, lo que se hace es aplicar un cuantificador al archivo original, con la idea de reducir el número de bits por muestra de los 16 que suelen ser normales en los ficheros de audio, a un valor mucho menor. Como sabemos, el hecho de realizar una cuantificación va a implicar necesariamente que se introduzca un determinado ruido de cuantificación (cuantos menos bits utilicemos por muestra, mayor será el ruido de cuantificación). Un codificador de audio lo que va a hacer es intentar controlar este ruido de cuantificación de modo que quede enmascarado por el sonido original, de forma que no sea audible. Si se hace correctamente, el archivo codificado y el original serían prácticamente indistinguibles desde un punto de vista perceptual.

Nosotros vamos a hacer algo mucho más sencillo, pero que permite observar el funcionamiento de un sistema de este tipo. En el archivo `Practica1b.py` se realiza la codificación de un archivo de audio de dos formas distintas:

- **En el dominio del tiempo**: Esto no es lo habitual (de hecho nunca se hace así), pero como primera aproximación lo que hacemos es simplemente recuantificar el archivo original, aplicarle un codificador Huffman y deshacer todo el proceso en recepción. Podéis observar el resultado final en el archivo *salida1.wav*.
- **En el dominio de la frecuencia**: En este caso lo que se hace es dividir el fichero de audio en tramas de N muestras, y para cada una de ellas se calcula una transformada al dominio de la frecuencia (usamos una transformada del coseno, DCT, en lugar de una transformada de Fourier porque devuelve valores reales, lo que facilita el trabajo). Una vez en el dominio de la frecuencia, se recuantifica cada una de las tramas, se le aplica un codificador Huffman, y el receptor deshace todo el proceso. La razón de trabajar de esta manera es que el proceso de enmascaramiento del oído humano también funciona en el domino de la frecuencia, por lo que al trabajar en este dominio podemos ajustar mucho mejor el funcionamiento del codificador a las características del sonido. En este caso podéis ver la salida del codificador en el archivo *salida2.wav*.

En realidad el proceso es algo más complejo. Si queréis profundizar algo más podéis echarle un ojo a este tutorial para haceros una idea: `https://ieeexplore.ieee.org/document/618009`. Podéis acceder a él desde la intranet de la UAH o utilizando la VPN.

4 Archivo de imagen

Este apartado tiene como idea complementar lo ya hecho anteriormente con un ejemplo de aplicación real como es la codificación de imágenes. En concreto vamos a pensar en una versión muy simplificada de un codificador JPEG que es lo que podéis encontrar en el archivo `Practica1c.py`.

JPEG se enmarca dentro de lo que se conocen como codificadores con pérdidas. Esto quiere decir que la imagen original nunca se va a poder volver a reconstruir a partir de la imagen en formato JPEG, ya que hay partes de la información que se van a eliminar. Igual que antes, esta información se elimina de forma que su impacto visual sobre la imagen sea el menor posible, de forma que, si todo va bien, la imagen JPEG y la imagen original sean prácticamente idénticas.

Para conseguir todo esto, el algoritmo JPEG hace lo siguiente con la imagen (vamos a pensar por simplicidad que tenemos una imagen en escala de grises):

1. Antes de nada, convertimos los valores de cada pixel de la imagen de formato *entero sin signo* a *entero con signo*. Para un caso como el que vamos a considerar, en el que la imagen viene codificada con 8 bits por muestra, esto implica simplemente restarle $2^7 = 128$ a cada muestra, de forma que los valores pasan de $[0, 255]$ a $[-128, 127]$.
2. Ahora procesamos la imagen en bloques de 8x8 píxeles.
3. Se calcula la Transformada del Coseno (DCT) para cada bloque. Como comentábamos antes, la DCT es una variante de la transformada de Fourier en la que las funciones base en lugar de ser exponenciales complejas son cosenos. Esto tiene algunas limitaciones que no vienen al caso, pero también tiene una gran ventaja, y es que los valores de la DCT son reales, por lo que reducimos a la mitad la información a codificar con respecto a una DFT normal.
4. Se cuantifica el bloque transformado utilizando una matriz de cuantificación. El estándar propone una, aunque luego cada fabricante puede utilizar una propia. Si llamamos Q a la matriz de cuantificación y X al bloque obtenido tras la DCT, el proceso es simplemente:
$$X_q = Round\left(\frac{X}{Q}\right) \tag{1}$$
5. Se aplica un proceso de codificación Huffman a la señal cuantificada

Una vez en el decodificador, se siguen los pasos inversos para cada bloque:

1. Hacemos la cuantificación inversa:
$$X_R = X_q \cdot Q \tag{2}$$
2. Calculamos la transformada inversa del coseno (IDCT).

En realidad lo que se hace es que se codifica de forma separada el primer pixel de cada bloque de 8x8 (que tiene un valor muy superior al resto de píxeles del bloque)[1]. El valor de estos píxeles iniciales de cada bloque se codifica de forma diferencial con el del anterior bloque, ahorrando así algún bit extra, y el resto de los píxeles sí que se codifican utilizando un código Huffman, ordenando los valores siguiendo un patron en diagonal.

5 Qué hay que entregar

- El archivo `FuncionesP1.py`.
- Un documento de texto en el que se responda a lo siguiente:

[1] Si accedéis desde la UAH, o con VPN, podéis echarle un ojo a este tutorial para haceros una idea: `https://ieeexplore.ieee.org/document/125072`

- ¿Qué diferencias aprecias entre la codificación Huffman y la LZW para el archivo de texto?
- Para el caso del archivo de audio, ¿qué diferencia existe entre trabajar en el dominio del tiempo o en el dominio transformado? ¿Qué sucede si aumentas o disminuyes el valor del escalón de cuantificación (está en la parte de Configuración del archivo `Practica1b.py`)?
- Lo mismo para el caso de la imagen. ¿Cómo afecta el valor del factor de calidad a los resultados?