Guion III: Compresión sin pérdida Codificación Aritmética

Información sobre la entrega de la práctica

Las prácticas se entregarán en un único fichero comprimido Practica03ApellidoNombre.zip. El fichero contendrá:

- Las funciones de Matlab a realizar en ficheros .m con los nombres de las funciones que se indiquen en el guion.
- Los trozos de código a realizar, que se entregarán todos en los pasos correspondientes de un único fichero .m llamado Practica03ApellidoNombre.m . Este fichero lo crearás modificando el fichero .m Practica03MolinaRafael.m en el servidor.
- Las discusiones y respuestas solicitadas en el guion se entregarán en un único fichero pdf. El nombre del fichero será Practica03ApellidoNombre.pdf. Lo construirás editando Practica03MolinaRafael.doc y salvándolo en formato pdf.

Codificación aritmética

En este apartado estudiaremos la codificación aritmética, un método que, al igual que la codificación Huffman, pretende reducir el número medio de bits requeridos para representar un símbolo pero que, a diferencia de ésta, permite representar símbolos con un número de bits fraccionario. Observa que para que el número de bits correspondientes a un símbolo sea fraccionario utilizando Huffman necesitamos codificar conjuntamente pares, tripletas, etc de símbolos, con los problemas de requerimiento adicional de espacio que esto genera.

Además del material que veremos en esta práctica te recomendamos que visites la codificación aritmética de Michael Dipperstein. Puedes usar el código en C de codificación aritmética de Michael Dipperstein para estudiar el funcionamiento práctico de la codificación aritmética mediante la realización de una serie de ejercicios o el código de las correspodientes funciones de Matlab que ahora veremos.

Por último antes de realizar algunos ejemplos observa que para saber cuándo hemos terminado de decodificar todos los datos podemos, bien podemos indicar el número de elementos que vamos a codificar o introducir un símbolo adicional (**EOF**) que sólo aparece una sola vez y que se incluye en la codificación de la secuencia. En la decodificación de la secuencia, la aparición de este símbolo indica que hemos terminado.

Visita también la página web del curso sobre Multimedia de David Marshall de la Universidad de Bristol http://www.cs.cf.ac.uk/Dave/Multimedia/ . En él encontrarás, en la sección Online Course Notes, pdfs de temas interesantes relacionados con el curso. En la sección BSc Multimedia (CM3106) Tutorial/Lab Class Notes, Exercises, example Code and Libraries encontrarás tutoriales y código en Matlab. Mira en particular la clase 7. Una copia local de Basic_compression.zip, que contiene ficheros .m de Matlab, la puedes encontrar en el material de la asignatura.

Paso 1

Vamos a comenzar con un ejemplo de Matlab. Consideremos una fuente con alfabeto {x, y, z}. La fuente ha generado la secuencia yzxxx que queremos codificar. Declaramos el alfabeto y la secuencia observada

```
close all; clear all; clc;
alf=['x' 'y' 'z'];
seqob=['y' 'z' 'x' 'z' 'z'];
```

Paso 2

Convertimos la secuencia observada a la secuencia de los índices correspondientes de la matriz del alfabeto. Los índices van del *1*, que corresponde al primer símbolo del alfabeto, a la *longitud del alfabeto*, es decir numel(alf), que corresponde al último símbolo del alfabeto

```
indseqob =[2 3 1 3 3];
```

Paso 3

Vamos a crear la secuencia de índices observados mediante órdenes de Matlab y no manualmente como hemos hecho en el paso 2. Inicializa la matriz que contendrá dichos índices (más tarde entenderás porqué creamos la matriz de tipo uint16)

```
indseqob=zeros(1,numel(seqob),'uint16');
```

Paso 4

Incluye en el paso 4 del fichero Practica03ApellidoNombre.m el código para convertir la secuencia seqob en índices del rango [1:numel(alf)]. Dichos índices los almacenarás en indseqob. Comprueba que indseqob contiene [2 3 1 3 3].

Paso 5

De un conjunto de entrenamiento hemos extraído las siguientes frecuencias de los símbolos. Éstas son, en el mismo orden que el alfabeto, [29 48 100]. Para codificar la secuencia que contiene los índices de los símbolos observados escribimos

```
counts=[29 48 100];
code=arithenco(indseqob,counts)
```

La salida es

code =

0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0

Lo que indica que hemos codificado la secuencia de 5 bytes indseqob en sólo 16 bits

Paso 6

A continuación queremos decodificar la secuencia que contiene la codificación aritmética de nuestra secuencia original. Simplemente escribimos

```
indseqdec=arithdeco(code,counts,numel(indseqob));
seqdecf=alf(indseqdec);
fprintf('¿Coinciden original y comprimido 1(S) 0 (N)?, %d\n',...
    isequal(seqob,seqdecf))
```

La salida es

¿Coinciden original y comprimido 1(S) 0 (N)?, 1

Paso 7

Vamos ahora a trabajar un poco los conceptos que hemos visto en teoría. Por simplicidad supondremos que las palabras de código son números enteros en el rango [1:256]. Ejecuta y entiende el siguiente trozo de código. Es importante que tengas claro cuál es el alfabeto

```
close all; clear all;
seqob=[1 1];
counts=[1 1];
code=arithenco(seqob,counts);
fprintf('Longitud de la secuencia codificada %d\n',...
    numel(code))
```

La salida es

Longitud de la secuencia codificada 5

Paso 8

Compara en el paso 8 del fichero Practica03ApellidoNombre.pdf esta longitud con la que la teoría nos dice que es una cota superior al número de bits necesarios.

Escribe tus respuestas aquí.

Para la codificación aritmética sabemos que tenemos certeza al dar un valor de etiqueta que tenga una cantidad de bits que viene dada por round_up(log2(1/(alto-bajo)))+1. Que en nuestro caso al ser las probabilidades P(1)=0.5 y P(2)=0.5, el intervalo final de una secuencia

de dos bits tendria como valor de (alto-bajo)=0.25, por lo que sería round_up(2)+1 = 3. Por tanto para enviar secuencias de dos bits necesitamos máximo 3 bits. Vemos como en nuestro caso estamos usando 5. Podemos mostrar el valor de code que produce nuestro algoritmo para todas las posibles combinaciones de 2 elementos y obtenemos el siguiente resultado:

[1 1]: 0 0 0 0 0 (0) Intervalo: [0,0.25)

[1 2]: 0 1 0 0 0 (0.25) Intervalo: [0.25,0.5)

[2 1]: 1 0 0 0 0 (0.5) Intervalo: [0.5,0.75)

[2 2]: 1 1 0 0 0 (0.75) Intervalo: [0.75,1)

Vemos como no es que esté dando un resultado incorrecto, es que el algoritmo de arithenco está dando un total de 3 cifras no significativas al final en todos los mensajes, por lo que realmente con 2 bits ya estamos mandando correctamente el intervalo en que se encuentra nuestro elemento.

Paso 9

Ejecuta ahora y entiende el siguiente trozo de código

```
close all; clear all;
seqob=[1 1];
counts=[50 50];
code=arithenco(seqob,counts);
fprintf('Longitud de la secuencia codificada %d\n',...
    numel(code))
```

La salida es

Longitud de la secuencia codificada 11

Paso 10

¿Por qué crees que ahora la longitud de la secuencia es 11 cuando las frecuencias [1 1] y [50 50] producen las mismas probabilidades?

Incluye la discusión en el paso 10 del fichero Practica03ApellidoNombre.pdf

Escribe tus respuestas aquí.

Nos encontramos con el mismo problema de antes, la función arithenco está añadiendo 0's al final de la etiqueta los cuales resultan absolutamente inútiles ya que no son cifras significativas y no aportan nada al resultado de la etiqueta. Ahondando en la función arithenco (click_derecho -> open arithenco) me fijo en la siguiente línea de código:

N = ceil(log2(total_count)) + 2;

Este valor N que se calcula haciendo la suma acumulada de todos los counts se utiliza para crear una estimación de la longitud final de la etiqueta y por tanto no emplea probabilidades ni decimales ni fracciones, utiliza el logaritmo en base 2 del número total de counts el cual varía entre [1 1] = 2 y [50 50] = 100. Probablemente sea un ajuste de eficiencia pensado para

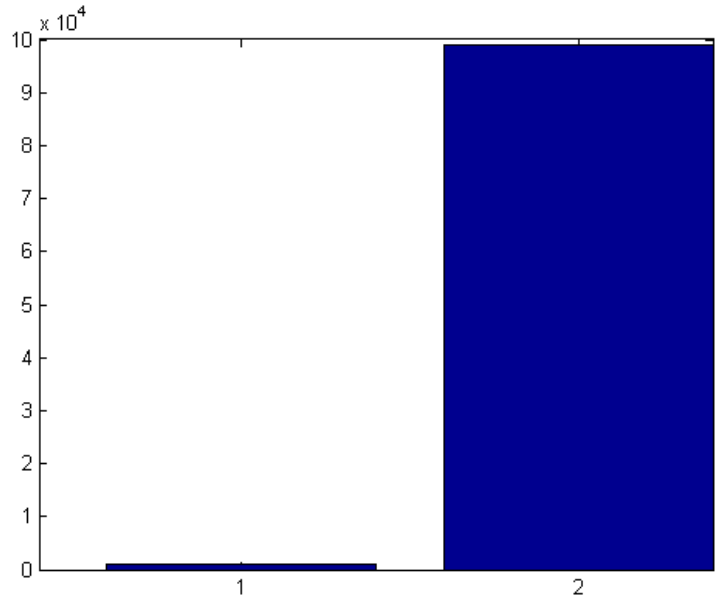
grandes volúmenes de datos pero que resulta ineficiente a la hora de tratar con pequeñas longitudes a codificar dentro de alfabetos con pocos símbolos y con counts de bajas cantidades.

Paso 11

Vamos ahora a analizar cómo funciona la codificación aritmética con distribuciones skew, es decir, distribuciones descompensadas. Comenzamos generando una secuencia de símbolos de un alfabeto con dos letras con probabilidades muy asimétricas. Esta secuencia nos servirá de conjunto de entrenamiento para calcular las frecuencias de cada símbolo. Observa que si no ha salido ningún 1 la secuencia de entrenamiento no nos servirá. A continuación construiremos su código aritmético

```
clear all;close all; clc;
maximo= 0.0;
minimo=0.0;
rng(0);
while maximo==minimo
seq=randsrc(1,100000,[1 2; 0.01 0.99]);
maximo=max(seq(:));
minimo=min(seq(:));
end
histo=histc(seq,[1 2]);
bar([1 2],histo);
```

El histograma que obtenemos es



Paso 12

Codificamos la secuencia observada, calculamos su longitud, la decodificamos y comprobamos que la secuencia original y la decodificada coinciden

```
code=arithenco(seq,histo);
fprintf('Longitud de la secuencia codificada %d\n',numel(code));
indseqdec=arithdeco(code,histo,numel(seq));
fprintf('¿Coinciden original y comprimido 1(S) 0 (N)?, %d\n',...
    isequal(seq,seqdec))
```

Las salidas son

```
Longitud de la secuencia codificada 8177
¿Coinciden original y comprimido 1(S) 0 (N)?, 1
```

Observa el tamaño en bits de la secuencia codificada y el tamaño original

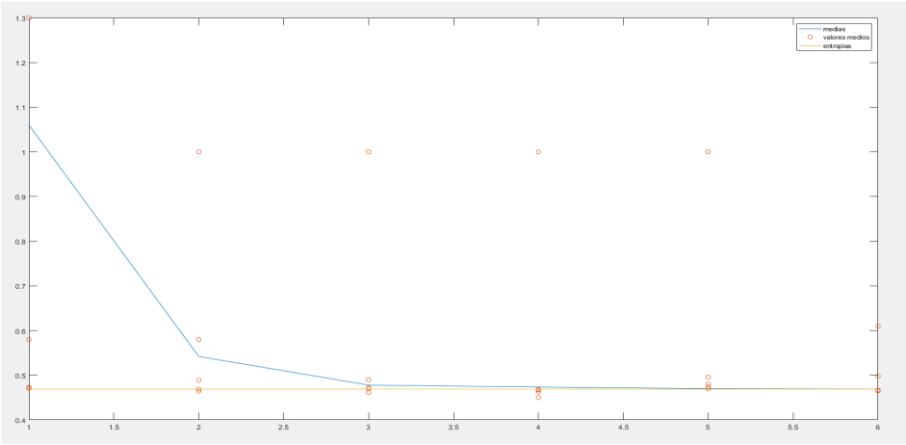
FC = 12.23

Paso 13

Realiza un estudio de cómo evoluciona el número de bits por símbolo cuando generamos 10^i, i=1,2,3,4,5,6 símbolos siguiendo el proceso del paso 11 con probabilidades pr(1)=0.1 y pr(2)=0.9. Para cada uno de los 6 casos, realiza 5 simulaciones distintas y calcula el número medio de bits por símbolo como una media de las 5 simulaciones. Dibuja los 30 valores obtenidos así como las medias de los 5 valores para las 10, 100, 1000, 10000, 100000 y 1000000 simulaciones. Dibuja también una línea con el valor de la entropía de la fuente. ¿Obtienes algún resultado que en principio parezca incorrecto entre la entropía de la fuente y los bits por símbolo de alguna simulación?, ¿Cuál sería la explicación?

Incluye el código en el paso 13 del fichero Practica03ApellidoNombre.m y la discusión en el paso 13 de Practica03ApellidoNombre.pdf.

Escribe tus respuestas aquí.



Podemos ver cómo sí que hay valores que superan aunque sea ligeramente el umbral de la entropía. Esto puede ocurrir ya que al ser casos experimentales las probabilidades de los dos elementos [1 2] no siempre serán exactamente [0.1 0.9] si no que pueden llegar a ser aún más dispares, en cuyo caso sabemos que la entropía disminuye y por tanto podemos obtener valores aún menores.

Paso 14

Como habrás podido comprobar empíricamente, la codificación aritmética es más eficiente cuanto mayor sea la longitud de la secuencia a codificar, es decir, el número medio de bits necesarios para representar cada símbolo se acerca más a la entropía cuantos más caracteres codificamos. Comprobaremos de nuevo empíricamente este hecho usando los ficheros de texto contenidos en el fichero `textobinario.zip`.

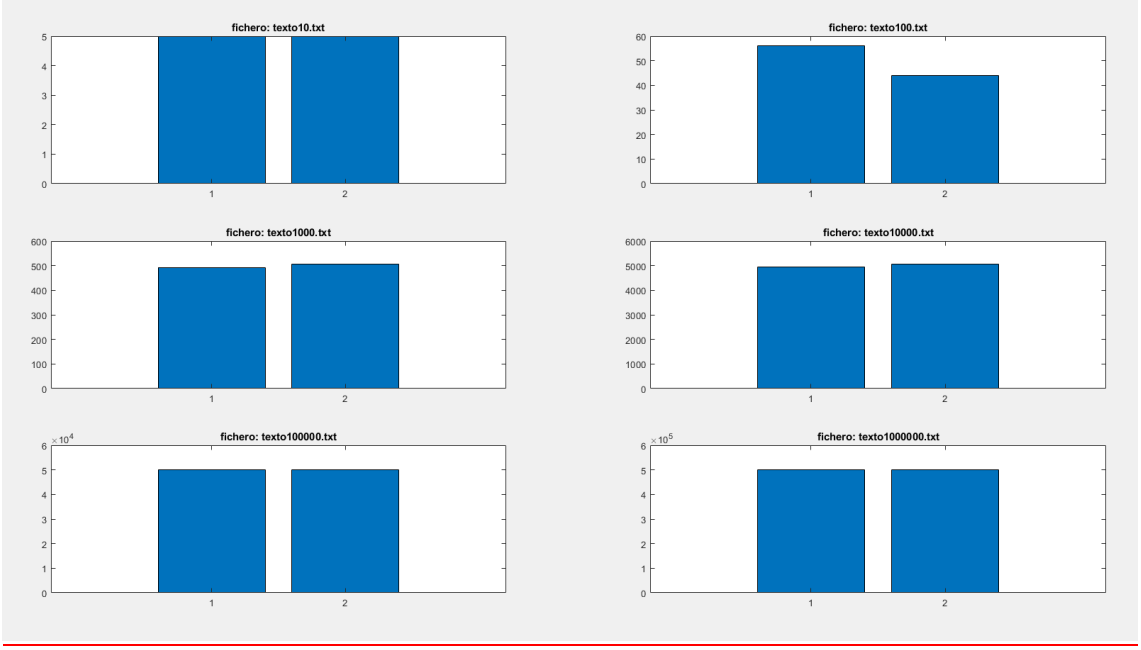
Los ficheros de texto en `textobinario.zip` contienen sólo dos caracteres, el carácter '0' y el carácter '1'. El nombre de cada fichero es `texto`*X*`.txt` donde *X* representa el número de caracteres en el fichero. Observa que los caracteres '0' y '1' vas a tener que codificarlos como 1 y 2.

Incluye en el paso 14 de Practica03ApellidoNombre.m el código para

1. codificar y decodificar cada uno de estos ficheros,
2. comprobar que la secuencia original y decodificada coinciden,
3. dibujar los histogramas de los símbolos en cada fichero.

Incluye los histograma, la tabla que contiene el número de bits por símbolo para cada fichero y las conclusiones que puedas extraer en el paso 14 de Practica03ApellidoNombre.pdf.

Escribe tus respuestas aquí.



Nº Caracteres	Bits/símbolo
10	1.600000e+00
100	1.070000e+00
1000	1.011000e+00
10000	1.001500e+00

100000	1.000180e+00
1000000	1.000021e+00

No se está consiguiendo ningún tipo de compresión, de hecho se está expandiendo el tamaño del fichero en comparación con el original. En teoría si las probabilidades son equitativas en el mejor caso se obtiene un factor de compresión de 1. Ya que se requiere la misma cantidad de bits para codificar la secuencia 1101010001 (10 bits) que para dar una etiqueta del intervalo (que siendo los probabilidades [0.5 0.5] seria el intervalo [0.8291015625, 0.830078125) cuya etiqueta con mínimos bits es precisamente el márgen inferior que se escribe como 0.1101010001, que es el mismo número que teníamos originalmente). Sin embargo, la propia función arithenco como hemos comprobado añade 0s adicionales. En este caso añade concretamente 5, lo cual lo hace peor que simplemente no codificando el archivo. En el caso de probabilidades algo más sesgadas la cosa mejora, pero en los archivos propuestos la cantidad de 0s y 1s son muy semejantes, y la cantidad de 0s añadidos (que con secuencias muy largas no tiene tanto peso como con el archivo de 10 elementos) compensa negativamente esa mejora de rendimiento, haciendo que finalmente el algoritmo codifique en más espacio del que teníamos originalmente.

Paso 15

Utiliza codificación aritmética para codificar cada uno de ficheros de texto constitucion española.txt, Fundacion e Imperio - Isaac Asimov.txt y Cinco semanas en globo - Julio Verne.txt (dentro de Prácticas - Datos - texto.zip).

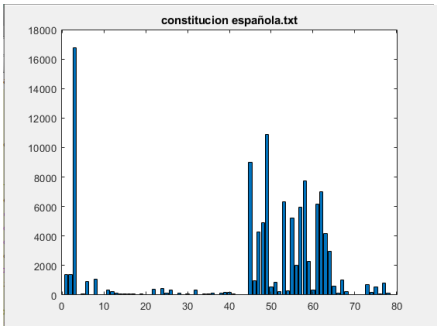
Incluye en el paso 15 de Practica03ApellidoNombre.m el código para

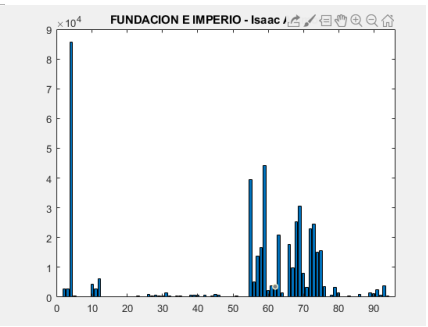
1. codificar y decodificar cada uno de estos ficheros,
2. comprobar que la secuencia original y decodificada coinciden,
3. dibujar los histogramas de los símbolos en cada fichero,
4. calcular el factor de compresión obtenida para cada uno de los ficheros,
5. calcular el número de bits por símbolo para cada fichero.

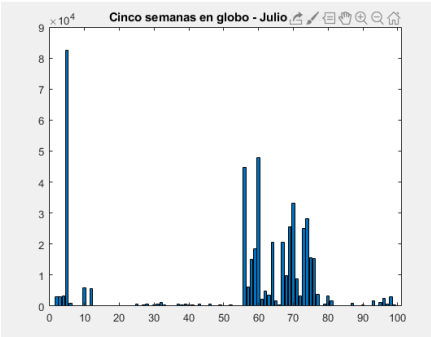
En el paso 15 de Practica03ApellidoNombre.pdf incluye

1. los histogramas,
2. completa las tablas adjuntas, para la codificación Huffman no incluyas el tamaño de la cabecera y
3. realiza una comparación crítica de los resultados obtenidos usando codificación Huffman y aritmética

Escribe tus respuestas aquí.







Fichero\Huffman	Tamaño fichero original	Factor de compresión	Bit/símbolo
Constitución española	112246 bytes	1.7711	4.516856e+00
Fundación e Imperio	461298 bytes	1.7826	4.487791e+00
Cinco semanas en globo	487020 bytes	1.7728	4.512739e+00

Fichero \aritmética	Tamaño fichero original	Factor de compresión	Bit/símbolo
Constitución española	112246 bytes	1,7825	4.488169e+00
Fundación e Imperio	461298 bytes	1.8006	4.442915e+00
Cinco semanas en globo	487020 bytes	1.7878	4.474773e+00

Comparación: No hay demasiada diferencia entre Huffman y la codificación aritmética. Al fin y al cabo Huffman es un sistema de compresión de muy alta eficiencia y puede ser óptimo incluso cuando las probabilidades son potencias negativas de 2. Sin embargo la diferencia de tamaño se incrementa mucho a favor de la codificación aritmética cuando tenemos en cuenta la cabecera ya que para alfabetos amplios Huffman requiere una cabecera de gran tamaño mientras que la codificación aritmética carece de este inconveniente.

Paso 16

Utiliza codificación aritmética sobre los ficheros de imágenes ptt1.pbm, ptt4.pbm, ptt8.pbm (dentro de material complementario - Datos para prácticas - imgs_binarias.zip) y camera.pgm, bird.pgm y bridge.pgm (dentro de Prácticas - Datos - imgs_grises.zip).

Incluye en el paso 16 de Practica03ApellidoNombre.m el código para

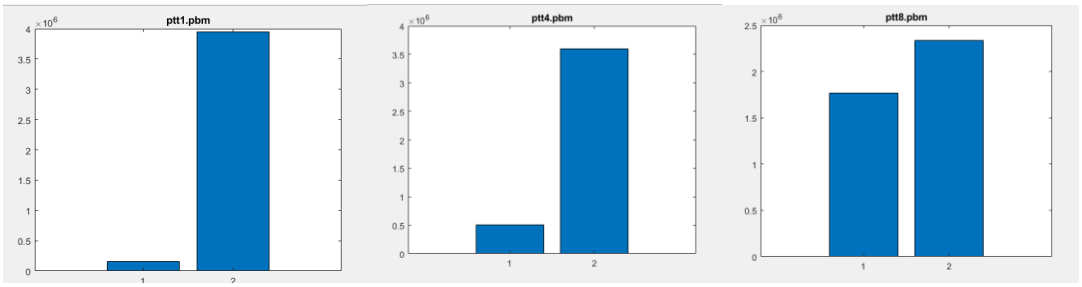
1. codificar y decodificar cada uno de estos ficheros,
2. comprobar que la secuencia original y decodificada coinciden,
3. dibujar los histogramas de los símbolos en cada fichero,
4. calcular el factor de compresión obtenida para cada uno de los ficheros,
5. calcular el número de bits por símbolo para cada fichero.

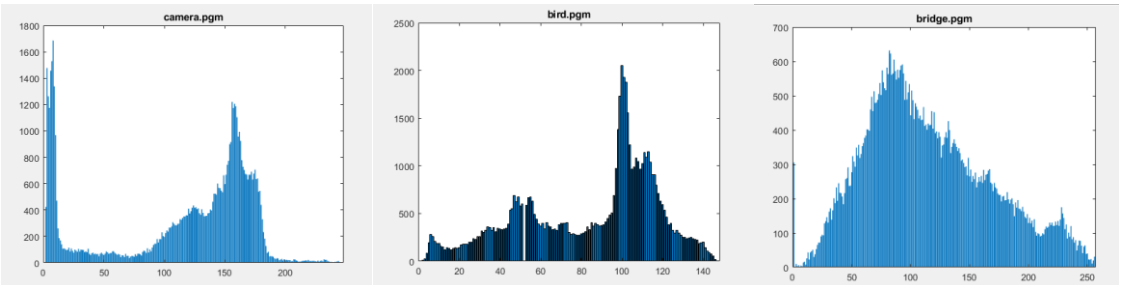
En el paso 16 de Practica03ApellidoNombre.pdf incluye

1. los histogramas,
2. completa las tablas adjuntas, para la codificación Huffman no incluyas el tamaño de la cabecera y
3. realiza una comparación crítica de los resultados obtenidos usando codificación Huffman y aritmética

Escribe tus respuestas aquí.

Los histogramas incluídos a continuación son tras eliminar del histograma los valores que no aparecen nunca y pasarlos al intervalo [1:num_elementos_no_nulos].





Fichero \Huffman	Tamaño fichero original	Factor de compresión	Bits/símbolo
ptt1.pbm	4105728 bits	1	1
ptt4.pbm	4105728 bits	1	1
ptt8.pbm	4105728 bits	1	1
camera.pgm	65551 bytes	1,1355	7.045293e+00
bird.pgm	65551 bytes	1,1761	6.802276e+00
bridge.pgm	65551 bytes	1,0398	7.693902e+00

Fichero \Aritmética	Tamaño fichero original	Factor de compresión	Bits/símbolo
ptt1.pbm	4105728 bits	4.2992	2.325670e-01
ptt4.pbm	4105728 bits	1.8480	5.411165e-01
ptt8.pbm	4105728 bits	1,0143	9.859199e-01
camera.pgm	65551 bytes	1,1411	7.010465e+00
bird.pgm	65551 bytes	1,1808	6.775442e+00
bridge.pgm	65551 bytes	1,0431	7.669097e+00

Podemos ver como la compresión aritmética supera en todos los ejemplos propuestos a la codificación de Huffman. Mientras que en los ejemplos de las imágenes con alfabetos entre [0:255] la diferencia no es demasiado grande, ésta se acentúa mucho para archivos binarios, ya que Huffman no tiene posibilidad de comprimir, al ser la mínima unidad de compresión el bit. Por tanto, si los datos originales ya se encontraban dados en bits, por mucho que las probabilidades estén sesgadas Huffman deja el archivo casi intacto (Sólo cambia los 1's por 0's y viceversa). Por el contrario la codificación aritmética sí que aprovecha, y muy eficientemente, ese sesgo en las probabilidades, obteniendo un factor de compresión de más de 4. Es decir, el fichero final ocupa menos de ¼ de lo que ocupaba el original.

Paso 17

No hemos discutido en clase qué cabecera debemos incluir en el fichero codificado para que el decodificador sea capaz de reconstruir el fichero original. Suponiendo que, como mucho, las letras del alfabeto son 256, ¿qué cabecera incluirías?. No olvides incluir las frecuencias o probabilidades si lo consideras necesario. Podemos incluir la longitud de la secuencia a decodificar, ¿habría alguna forma de no tener que incluir el número de símbolos a decodificar?.

Incluye la discusión en el paso 17 de Practica03ApellidoNombre.pdf.

Escribe tus respuestas aquí.

En primer lugar es imprescindible comentar que, a menos que se utilice un sistema alternativo, que probablemente los haya, es imprescindible enviar la longitud de la secuencia original. Pongamos el siguiente caso, disponemos de 20 símbolos [1:20] con probabilidades de 1/20 cada uno, sin embargo queremos codificar la secuencia 1111111111, ésta estará en el

intervalo [0, 0.algo). Una etiqueta posible y la que nuestro algoritmo enviaría sería la 0. Pero en este caso el decodificador, sólo con la etiqueta, no es capaz de diferenciar entre las secuencias [1, 11, 111, 1111, …] ya que la etiqueta para todas ellas sería la misma, el 0. Por supuesto también es necesaria una tabla de frecuencias/probabilidades para que el decodificador sea capaz de reconstruir la recta [0,1] y segmentarla debidamente.

Un método alternativo, como comentaba anteriormente, sería forzar a que todo fichero tuviera un símbolo indicando el final del fichero (con frecuencia 1 evidentemente, probabilidad 1/(tamaño_fichero+1)) y añadirlo en los calculos de frecuencias/probabilidades como si fuera un símbolo más. De esta manera cuando decodifiquemos un valor que se encuentra en el segmento de la recta que corresponde a éste símbolo sabemos que hemos terminado de decodificar, mientras tanto continuamos aunque esto conlleve decodificar numerosas veces el mismo símbolo.