Aprendizaje automático

Práctica 7: Agrupamiento (Clustering)

GAJAN Antoine (894825)

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Índice

Introducción

Como parte de la asignatura “Aprendizaje automático", los estudiantes se familiarizarán con los conceptos de inteligencia artificial. Más concretamente, durante el cuatrimestre se estudiarán las bases del aprendizaje supervisado y no supervisado.

Durante esta séptima práctica, los estudiantes se familiarizarán con las nociones de agrupamiento y clustering, conceptos muy utilizados en la inteligencia artificial. Las nociones adquiridas en las clases magistrales se aplicarán aquí para comprender las ventajas y desventajas del agrupamiento con el algoritmo K-Medias.

Esta memoria pondrá de relieve el proceso emprendido para responder a las preguntas, y producirá una reflexión personal sobre los resultados obtenidos.

1. Estudio previo

Durante este estudio previo de la práctica, me tomé el tiempo de revisar los algoritmos propuestos en el curso magistral, entre ellos el de K Means. En particular, investigué las diferentes formas de generar clústeres aleatorios al principio del algoritmo, cómo asignar un clúster a un dato y cómo actualizar los centroides. Revisar estas nociones me ha permitido realizar la práctica en las mejores condiciones para poder interpretar mejor los resultados obtenidos.

2. Memoria de la práctica

2.1. Implementación de KMeans

Utilizando los archivos y las funciones auxiliares que ofrece la declaración, podemos crear fácilmente el algoritmo KMeans.

2.1.1. Centroides iniciales

En primer lugar, tenemos que inicializar los K centroides iniciales. Para ello, como hemos aplicado la metodología vista en curso. Vamos a elegir K datos aleatorios y distintos, y considerar que cada uno de estos datos es un centroide. Hicimos esta elección porque vimos que mejora el rendimiento del algoritmo después. Nos aseguramos de que cada clúster contenga al menos un dato.

Con el fin de lograr esto en Matlab, he decidido proponer el siguiente algoritmo:

Algorithm 1 initialize_mu0(D, K)

{Initializacion mu0}

$mu0 \leftarrow []$

$nb\_clusters \leftarrow 0$

{Continuamos hasta que tengamos K valores}

while $nb\_clusters < K$ **do**

{Generar valor aleatorio entre 1 y numero de lineas de D}

  $index \leftarrow random(1, size(D))$

  $data\_selected \leftarrow D(i, :)$

  if $data\_selected$ not in $mu0$ **then**

    $mu0 = [mu0; data\_selected]$

    $nb\_clusters \leftarrow nb\_clusters + 1$

  end if

end while

return mu0

2.1.2. Actualización de los centroides

Un segundo paso importante en el algoritmo KMeans es poder actualizar el valor de los centroides. Para ello, como hemos visto en el curso, la idea es asignar al centroide i, la media de los valores clasificados como de clase i.

Así, para realizar esta función de actualización, he propuesto el siguiente algoritmo:

Algorithm 2 update_centroids(D, c)

$munew \leftarrow []$

$K \leftarrow max(C)$

for $k \leftarrow 1$ to K **do**

$D\_good \leftarrow D(c == k, :)$

$munew \leftarrow [munew; mean(D\_good)]$

end for

return munew

2.1.3. Actualización de los clústeres

Después de calcular el valor de mu, es necesario actualizar los clústeres. Este término debe entenderse en el sentido de actualizar la clase a la que pertenece cada uno de los datos, a partir de mu. Esta clase i corresponde al índice que permitió minimizar la distancia euclidiana al cuadrado entre D y mu.

Así, gracias a las funciones Matlab, podemos ofrecer el siguiente algoritmo. Me tomé el tiempo en un primer momento para realizarlo manualmente para comprender el funcionamiento, luego me aseguré de optimizarlo al máximo para reducir la complejidad temporal de la función. En efecto, me he visto obligado a optimizarlo para obtener un resultado en un plazo razonable en la segunda parte de la práctica.

Algorithm 3 update_clusters(D, mu)

{Calcular distancias al cuadrado entre cada dato de D y cada centroid de mu}

$distances \leftarrow pdist2(D, mu, 'squaredeuclidean')$

{Obtener el indice correspondiente al minimo de la distancia para cada linea}

$[\ , Z] = min(distances, [], 2)$

return Z

2.1.4. Implementación de KMeans con funciones auxiliares

Gracias a las funciones auxiliares realizadas anteriormente, ahora somos capaces de proporcionar un código para el algoritmo KMeans que es lo más fiel posible a la mencionada en el curso.

Así, podemos proponer el siguiente algoritmo:

Algorithm 4 Kmeans(D, mu0)

{Initializacion variables}

$m \leftarrow size(D, 1)$

$n \leftarrow zeros(m, 1)$

$mu \leftarrow mu0$

{Continuamos hasta que c no combie}

$cond \leftarrow true$

while *cond* **do**

$previous\_c \leftarrow c$

{Actualizacion clusters y centroides}

$c \leftarrow update\_clusters(D, mu)$

$mu \leftarrow update\_centroids(D, c)$

{Actualizacion condicion}

$cond \leftarrow c! = previous\_c$

end while

return mu, c

2.1.5. Mejora hacia la convergencia

Aunque no se pide explícitamente en el enunciado, deseé implementar el algoritmo mejorado para tender hacia una mejor convergencia. Por lo tanto, utilizando las propiedades de la función de distorsión vista en clase, repetiremos N veces el algoritmo KMeans para mantener el modelo que minimiza la función de distorsión. Así, el modelo propuesto será lo más justo posible y podremos movilizarlo posteriormente en imágenes para ver los beneficios de KMeans.

Dado que la función de distorsión es relativamente fácil de implementar y la optimización consiste solo en iterar N veces con un bucle for, no voy a detallar el algoritmo aquí, pero está presente en el archivo optimized_KMeans. m

2.2. Aplicación de Kmeans a un caso concreto

En esta segunda parte de la práctica, vamos a poder utilizar la función previamente implementada para probar su eficacia y ver sus ventajas y desventajas.

En un primer momento, como se indica en el enunciado, vamos a aplicar el algoritmo KMeans a la imagen del loro para agrupar los píxeles (tabla de 3 valores, correspondiente al nivel de rojo, verde y azul) en 16 categorías.

2.2.1. Imagen del loro pequeño

Para poder trabajar en esta imagen, es necesario extraer sus propiedades interesantes. Por lo tanto, primero almacenaremos todos los píxeles de la imagen como una matriz de M filas por 3 columnas. Para ello, podemos utilizar las funciones Matlab y el código propuesto como ejemplo en el archivo imageQuantization.m.

Figura 1: Imagen original del loro pequeño

Con el fin de comprender mejor la imagen a tratar, me tomé el tiempo de hacer un trazado 3D de los píxeles para visualizar si se agrupó o no. El resultado fue el siguiente:

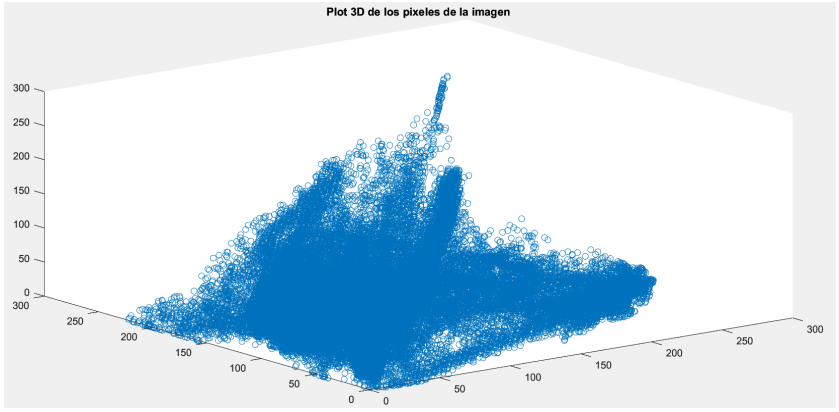

Figura 2: Representación de los píxeles en el espacio 3D de la imagen del loro

Después de observar los puntos del píxel en las 3 dimensiones, me di cuenta de que podríamos agrupar todos estos píxeles en un número muy pequeño de clústeres.

Una vez que la imagen se convierte en matriz y la función Kmeans se llama en la matriz, podemos reconstruir la imagen con el código propuesto. Obtenemos el siguiente resultado (con K = 16, valor propuesto inicialmente en el archivo Matlab):

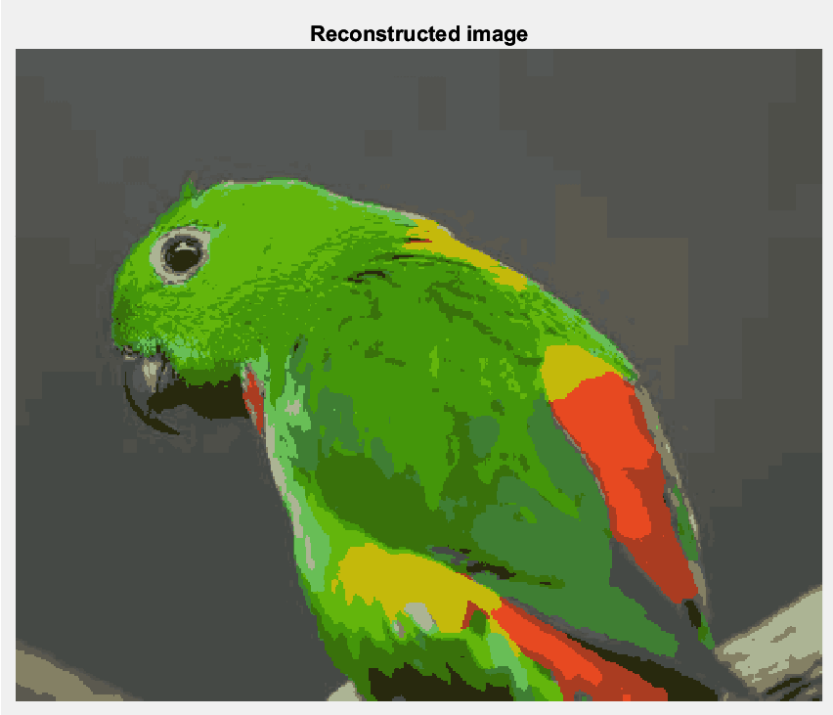

Figura 3: Imagen reconstruida del loro pequeño con 16 colores

La reconstrucción es increíble. De hecho, obtenemos una imagen reconstruida muy similar a la original, mientras que la imagen reconstruida contiene solo 16 colores.

Mantener solo 16 colores en esta imagen no afecta la calidad de la imagen. Deduzco que KMeans se puede utilizar para comprimir una imagen para ahorrar espacio de memoria. De hecho, mientras que la imagen inicial podía contener hasta $255^3 = 16\ 581\ 375$ colores, con solo 16 colores logramos mantener la nitidez de la imagen.

Sin embargo, esto me lleva a reflexionar sobre varios ejes. De hecho, aquí, K = 16 parece ideal, ya que la imagen inicial contenía relativamente pocos colores. ¿Qué pasaría con la imagen de un paisaje con muchos colores? ¿Cómo elegir K para la compresión? Estos dos puntos se abordarán en el resto de esta memoria.

2.2.2. Elección de K

Hemos visto que K = 16 parecía ser relevante, ya que la imagen del loro parece contener, a ojos humanos, menos de 16 colores en total. Sin embargo, si tenemos muchas imágenes para procesar, no podemos permitirnos mirarlas una a una y elegir el valor de K como mejor nos parezca.

Por lo tanto, para elegir K, decidí proceder de manera similar a lo que habíamos visto en curso. En particular, probé el método del codo. Este método consiste en estudiar la función de distorsión en función del número K de clústeres. Según el método, hay que observar el punto de la curva donde la disminución de la distorsión se vuelve más lenta (parece un codo). Este punto es el número óptimo de clústeres.

Al trazar la distorsión según el número K, obtuve el siguiente resultado:

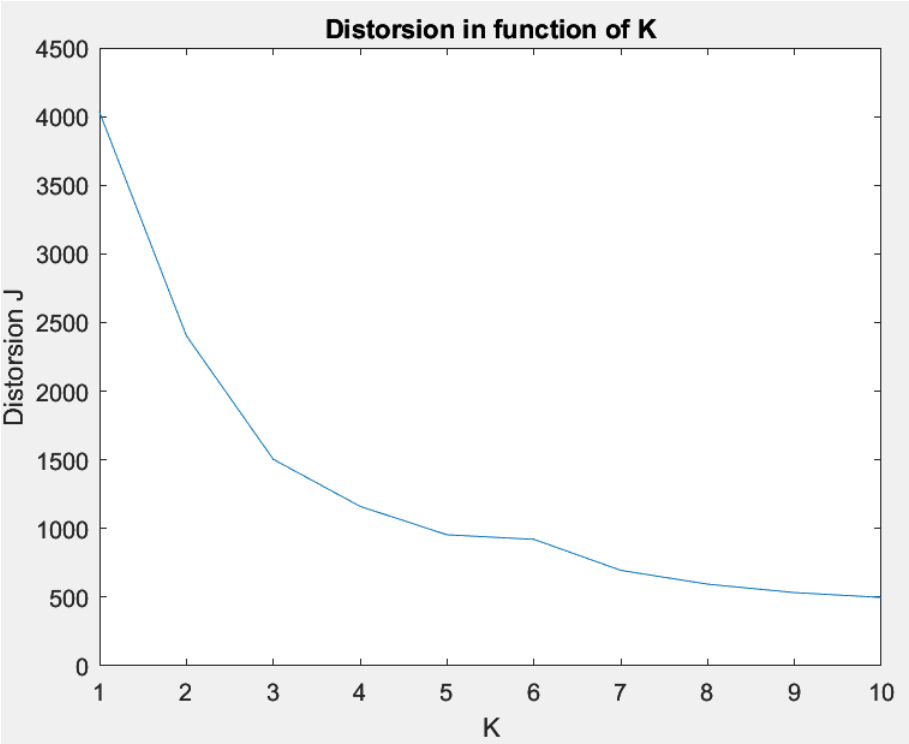

Figura 4: Evolución de la distorsión en función de K con la imagen del loro

Como hemos visto en curso, el valor de K preferido es aquel en el que observamos una desaceleración en la disminución de la curva (punto de inflexión). Aquí, este punto aparece cuando $K = 3$.

Así, en caso de que $K = 3$, obtenemos la siguiente imagen reconstruida. Esta, aunque bicolor, también es increíble. Hemos logrado, con solo 3 colores, reconstruir una imagen del loro bastante fiel a la imagen original.

Figura 5: Imagen reconstruida del loro con 3 colores

Por lo tanto, el método del codo nos permitió elegir sabiamente el valor del número de clústeres K. En particular, Me di cuenta de que elegir sabiamente el valor K mejora la compresión de la imagen para que ocupe menos espacio en la memoria (ya que tiene menos color). De hecho, con la nueva imagen, ahora podríamos almacenar la imagen comprimida con el siguiente formalismo. Cada píxel podría representarse como un número binario (0 o 1 o 2) en lugar de 3 valores entre 0 y 255 con el formalismo RGB.

Así podemos calcular la relación de compresión.

La imagen inicial consta de 480 * 388 píxeles inicialmente. Por lo tanto, debe almacenar 480 * 388 * 3 = 558 726 valores.

Con la imagen reconstruida según nuestro formalismo, para almacenarla tenemos que almacenar 480 * 388 = 186 240 bits. Tenemos que almacenar la matriz de coincidencia de colores en paralelo. Tenemos 3 colores RGB que almacenar, así que 9 bits. Por lo tanto, para almacenar la imagen comprimida, nos llevó 186 249 bits.

Por lo tanto, la tasa de compresión (Tamaño del archivo sin comprimir/ Tamaño del archivo comprimido) vale 3.

2.3. Pruebas con otras imagenes

2.3.1. Con una imagen más compleja

En el deseo de comprobar la veracidad de los resultados propuestos anteriormente, ejecuté mi código con diferentes imágenes más o menos complejas. La imagen que he elegido reflejar en este párrafo es la imagen sunset.jpg, que es muy grande y que contiene muchos colores. Obviamente, es posible probar el código con otras imágenes si lo desea.

Unas de las imagenes que he elegido es esta:

Figura 6: Paisaje con numerosos colores

Para procesar esta imagen, he procedido de la misma manera que antes. En otras palabras, reemplacé smallparrot.jpg por sunset.jpg en el archivo Matlab y lo ejecuté.

En particular, con el método del codo, obtuve el siguiente gráfico:

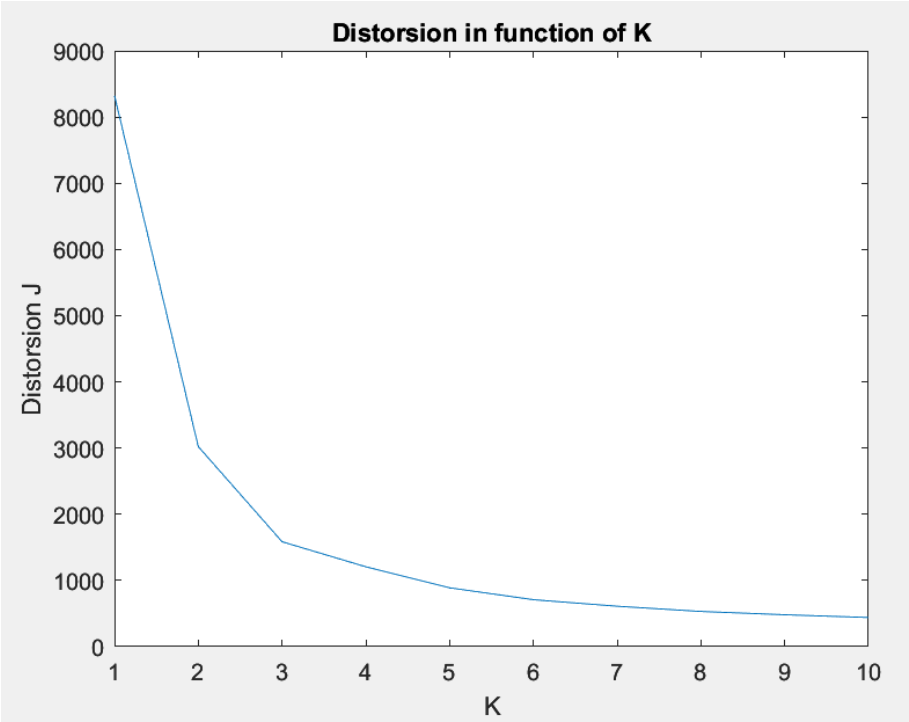

Figura 7: Evolución de la distorsión en función de K con la imagen del paisaje

Observamos que aunque la imagen parece estar compuesta de muchos más colores que la del loro, el método del codo nos indica la presencia del codo en el nivel de K = 3. Por lo tanto, elegiremos para optimizar la distorsión el valor K = 3, esto nos permite mejorar la compresión de nuestra imagen.

Con K = 3, obtenemos la siguiente imagen reconstituida:

Figura 8: Imagen reconstruida del paisaje con 3 colores

El resultado es de nuevo significativamente increíble. De hecho, con solo 3 colores, hemos logrado conservar las propiedades de la imagen original. Esto muestra una vez más las ventajas de KMeans para la compresión de imágenes. De hecho, en lugar de utilizar 3 variables para modelar el nivel de rojo, verde y azul, podríamos utilizar para representar un píxel una sola variable que tiene como valores 0, 1 o 2.

Así podemos calcular la relación de compresión.

La imagen inicial se compone de 770 * 512 píxeles inicialmente. Por lo tanto, hay que almacenar 770 * 512 * 3 = 1 182 720 valores.

Con la imagen reconstruida según nuestro formalismo, para almacenarla tenemos que almacenar 770 * 512 = 394 240 bits. Tenemos que almacenar la matriz de coincidencia de colores en paralelo. Tenemos 3 colores RGB que almacenar, así que 9 bits. Por lo tanto, para almacenar la imagen comprimida, nos llevó 394 249 bits.

Por lo tanto, la tasa de compresión (Tamaño del archivo sin comprimir/ Tamaño del archivo comprimido) vale 3.

2.3.2. Con una imagen de colores más vivos

Elegí esta segunda imagen del paisaje florecido (flowers.png) por varias razones. Esta imagen me parece más difícil de reconstruir sin perder demasiada información. De hecho, en primer plano, tenemos flores de diferentes colores. Así que me pregunto qué resultado obtendré.

Figura 9: Imagen de flores con colores más vivos

Al aplicar inicialmente el método del codo, decidí elegir K = 4.

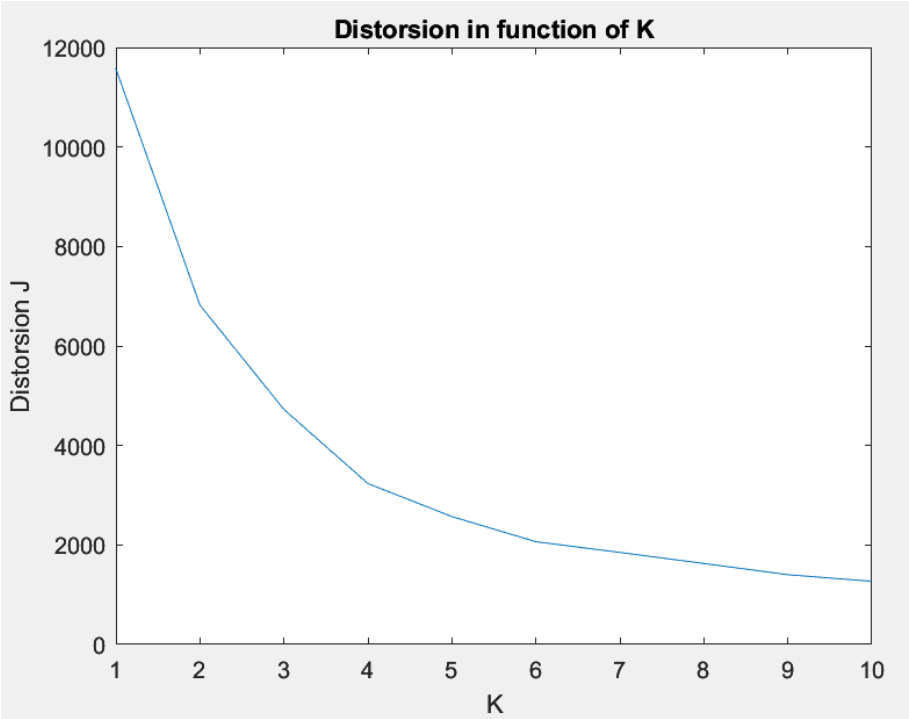

Figura 10: Evolución de la distorsión en función de K con la imagen del paisaje florecido

Aplicando KMeans, pude reconstruir la imagen y obtener el siguiente resultado:

Figura 11: Imagen reconstruida del paisaje florecido con 4 colores

Como era de esperar, podemos detectar que es un paisaje de flores. Sin embargo, observamos que el color rojo brillante de algunas flores no sobresale de la imagen y se confunde con el color del fondo.

2.3.3. Con una imagen sencilla y bicolor

Habiendo sido convencido por los resultados con imágenes complejas de grandes dimensiones, ahora he optado por procesar una imagen muy simple que es bicolor. Esta prueba me permitirá comprobar si KMeans tiene buenas propiedades de compresión en todo tipo de imágenes o solo en imágenes con un gran número de colores.

Figura 12: Imagen bicolor

Con las imágenes anteriores, la representación de píxeles en el espacio 3D no nos permitía deducir fácilmente el número de clústeres. Aquí, como la imagen es muy simple, notamos la presencia de 2 clústeres principales:

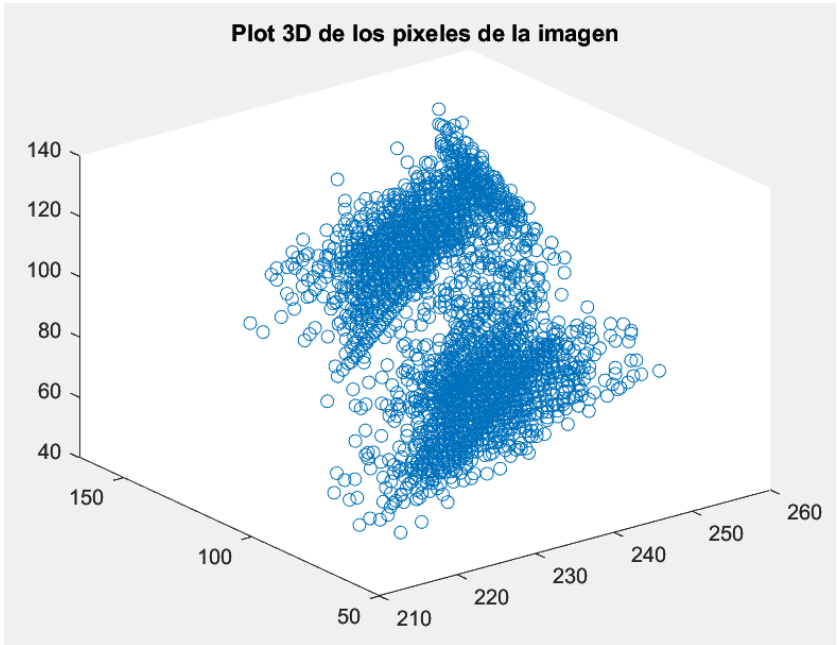

Figura 13: Representación de los píxeles en el espacio 3D de la imagen del loro

Con un procedimiento análogo al anterior, conseguí que había que elegir K = 2 (lo que es tranquilizador, ya que la imagen es bicolor).

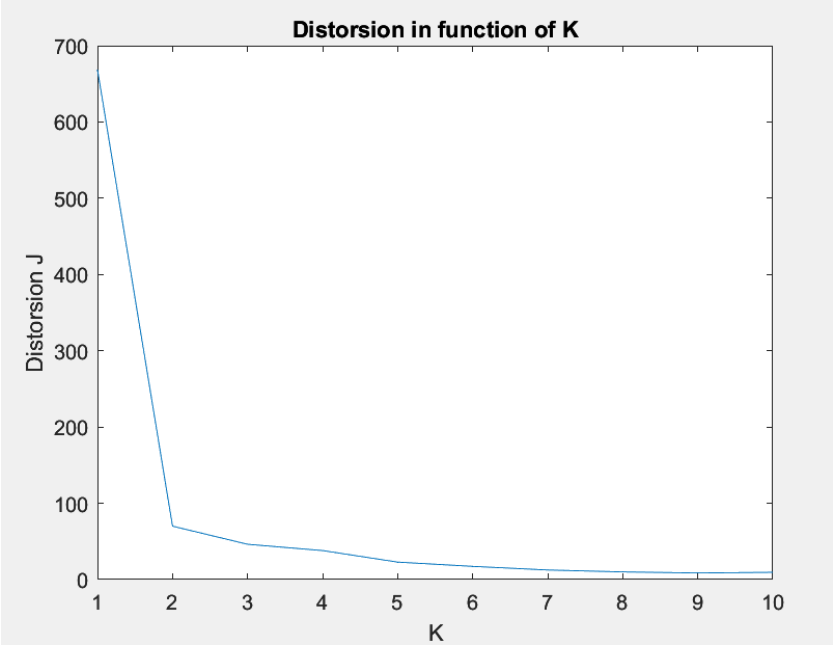

Figura 14: Evolución de la distorsión en función de K con la imagen de la imagen bicolor

Finalmente, conseguí la siguiente imagen reconstruida:

Figura 15: Imagen reconstruida de la imagen bicolor

No es de extrañar, la imagen es muy similar a la original. Deducimos que la compresión es muy potente. De hecho, al reducir a solo 2 colores, tenemos como

antes el aspecto general de la figura (podemos deducir lo que representa), pero además tenemos los colores correctos.

Sin embargo, si queremos ser quisquillosos, podemos notar algunos errores realizados al agrupar para los píxeles situados en la parte inferior derecha de la imagen que se han asociado con el clúster equivocado (color incorrecto).

Conclusión

Para concluir, durante esta práctica hemos implementado el algoritmo KMeans y visto sus beneficios en la compresión de imagen (reducción de 3 veces el tamaño de la imagen almacenada).

Como hemos visto, es un algoritmo muy fácil de entender e implementar. Funciona bien en datos de pequeño a mediano tamaño. Sin embargo, hay que tener en cuenta que su convergencia no es óptima, de ahí la necesidad de realizar varias iteraciones con diferentes centroides intiales. También pide conocer de antemano el número de clusters, algo que no se conoce necesariamente, de ahí la necesidad de utilizar el método del codo.

Por último, sería interesante ver los beneficios de KMeans en otras áreas de aplicación. En particular, he visto que es muy utilizado por empresas de redes sociales como Facebook para agrupar a los usuarios en función de sus centros de interés. También, KMeans se utiliza para detectar fraudes en transacciones bancarias.

Índice de figuras