Repte 7

Alumne: Martí Caixal i Joaniquet
NIU: 1563587, data: 28/04/2022

1 Introducció

Al repte 7 es demana implementar l'algorisme *k-means* amb l'objectiu d'utilitzar-lo per segmentar imatges. No només permetrà segmentar, sinó que al reduir el nombre de colors, també es comprimeix la mida.

Per realitzar vàries proves, es fan servir diferents espais de característiques:

- RGB: 3 canals de colors (red, green, blue) amb valors entre 0 i 255
- YUV: 3 canals (1 de luma, 2 de chominància) amb valors entre 0 i 255
- HSV: 3 canals (Hue, Saturation, Value) amb valors entre 0 i 1

El codi sencer de l'algorisme es pot trobar al final del document.

2 Imatges i preprocessament

Tot i demanar-se només una fotografia, s'ha utilitzat dos per aconseguir trobar problemes en casos determinats i diferents resultats.

A continuació es mostren les imatges utilitzades:

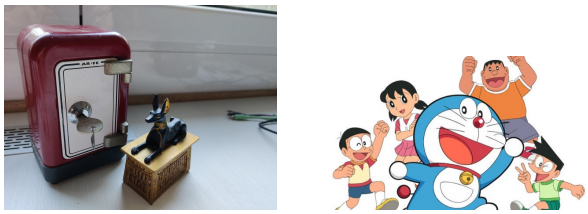

Figura 1: Imatges originals

Per poder tenir uns temps d'execució acceptable, s'ha hagut de reduir la mida de les imatges a aproximadament 400x300 píxels.

3 Algorisme K-means

K-means és un algorisme no supervisat de clustering molt utilitzat. La idea principal és agrupar un seguit de mostres o elements a una classe en concret. Al ser no supervisat, en cap moment es pot saber quines són les classes que trobarà.

Es basa en 4 apartats principals:

- Inicialització: S'escolleixen K colors aleatoris de la imatge. Aquests actuaran com a centroides inicials per començar l'execució.
- Assignació: Per cada píxel de la imatge, assignar-lo al centroide més proper. Per els espais RGB i YUV s'aconsegueix un bon resultat utilitzant la distància euclideana. En canvi, el HSV presenta problemes si s'utilitza aquesta distància. Més endavant es veurà com afecta.
- Nou centre: Calcular la nova posició de cada centroide tenint en compte la posició dels punts assignats a ell.
- Repetir: Cal seguir de nou aquest bucle fins a convergir. És a dir, fins que els centroides ja no canviïn de posició. A vegades no convergeixen mai, així que és comú definir un nombre màxim d'iteracions.

4 Resultats

Les proves s'han realitzat amb el valors K = 2, 4, 8, 16. Així doncs, cada imatge resultant acabarà tenint tants colors com el seu valor K

Per la primera imatge (caixa vermella i figura) es mostren tots els resultats obtinguts. Per la segona es mostren els més interessant per tal de no posar tantes imatges.

4.1 Caixa i figura

Mirant des de la figura 2 fins la figura 5, es pot veure l'evolució dels resultats de l'algorisme a mesura que augmenta el valor de K.

Per K=2 i k=4, si bé ja permet veure el contingut de la imatge, el valor encara és massa petit per tenir una bon resultat i poder distingir els colors.

Amb K=8 sí que es pot apreciar un canvi. Tant RGB com YUV defineixen bé els contorns dels objectes però els colors mostrats estan una mica apagats. Amb HSV passa el contrari, els contorns no estan ben definits, però els colors són més vius i fidels a la imatge original.

Amb K=16 es nota encara més els efectes comentats per el valor anterior de K.

El millor resultat a simple vista sembla ser el YUV. Tant els detalls com els contorns estan ben definits i és capaç de donar colors molt vius a les zones amb reflexes i a la vegada mantenir alguns colors com el vermell de la caixa bastant fidels a l'original. El HSV, vist des de lluny, pot semblar també un bon resultat i amb millors colors, però algunes textures queden planes i sense detall.

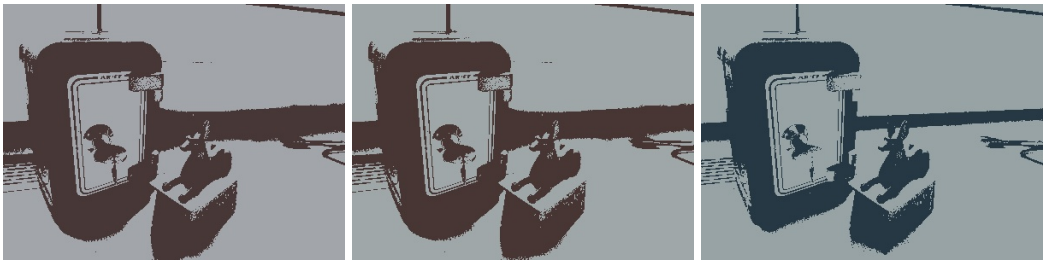

Figura 2: K = 2. Ordre RGB, YUV, HSV

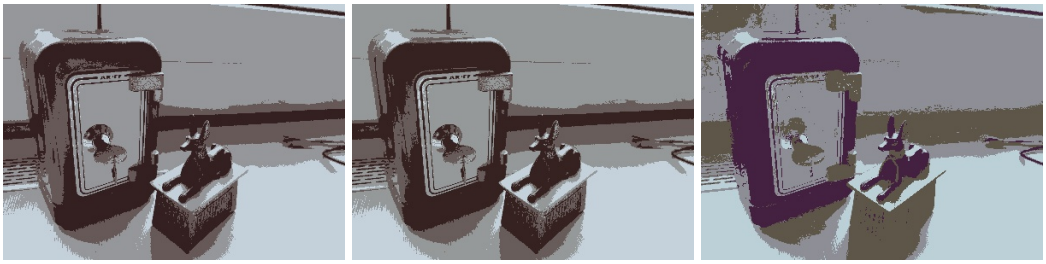

Figura 3: K = 4. Ordre RGB, YUV, HSV

Figura 4: K = 8. Ordre RGB, YUV, HSV

Figura 5: K = 16. Ordre RGB, YUV, HSV

4.2 Doraemon

La imatge del Doreamon es diferencia de l'anterior per ser d'animació 2D. Així doncs, al no tenir profunditat, la llum no canvia les tonalitats i una mateixa superfície té un sol color. La imatge original, per tant, ja té una baixa quantitat de colors únics.

A la figura 7 es veu que l'espai RGB i YUV, tot i tenir K=4, ja representen millor la imatge original que no pas l'espai HSV amb K=8. La raó molt segurament és l'ús de distància euclideana amb els 3 espais. Com es veu a la figura 6, els colors a HSV estan definits de manera semblant als graus d'una circumferència. Un vermell tirant a granat té un valors molt alt, mentre que un vermell tirant a taronja té un valor molt baix. Tot i això, són colors molt semblants. Si ens fixem de nou amb la figura 7, la zona de l'interior de la boca, on originalment és granat, l'espai HSV agafa el color Blau. El mateix passa per algunes peces de roba.

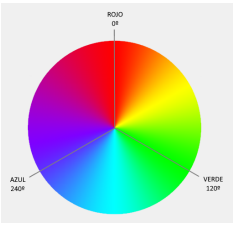

Figura 6: Colors a HSV

Figura 7: RGB i YUV amb K=4. HSV amb K=8

Anant a buscar els millor resultats per K=16, de nou l'espai YUV aconsegueix la millor representació a simple vista (veure figura 8). L'espai RGB manté algunes zones amb un color que no correspon (samarretes) i altres zones estan pigmentades amb píxels d'un altre color.

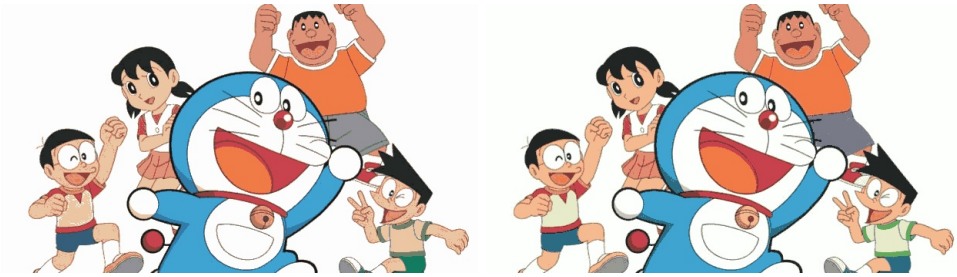

Figura 8: K = 16. Ordre RGB, YUV

5 Problemes trobats

La inicialització dels centroides ha presentat problemes amb la imatge del Doraemon (veure figura 9). Inicialment, per inicialitzar els centroids, s'agafava el valor de píxels de manera aleatòria . Amb la primera imatge, al tenir molts colors únics, la sort ha permès que tots els centroides inicials fossin diferents. La segona imatge, en canvi, té moltes zones d'un mateix color blanc i la mala inicialització ha presentat aquest problema.

No és res que no es pugui solucionar mitjançant `np.unique()`, però sí que és quelcom a tenir en compte.

Figura 9: Resultat incorrecte

Annex

Visualització centroides

A continuació es mostra una visualització dels culsters que es troben a l'espai RGB per diferents K de la imatge Doraemon.

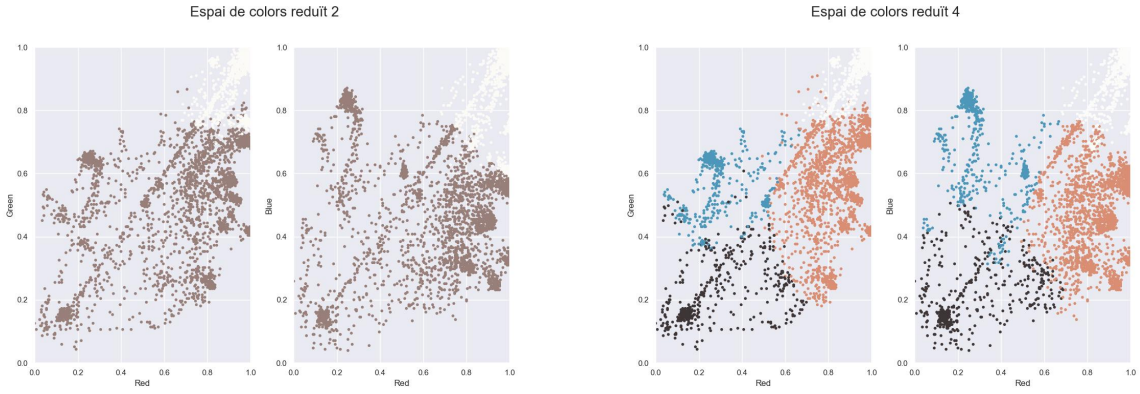

Figura 10: Ordre de K: 2, 4

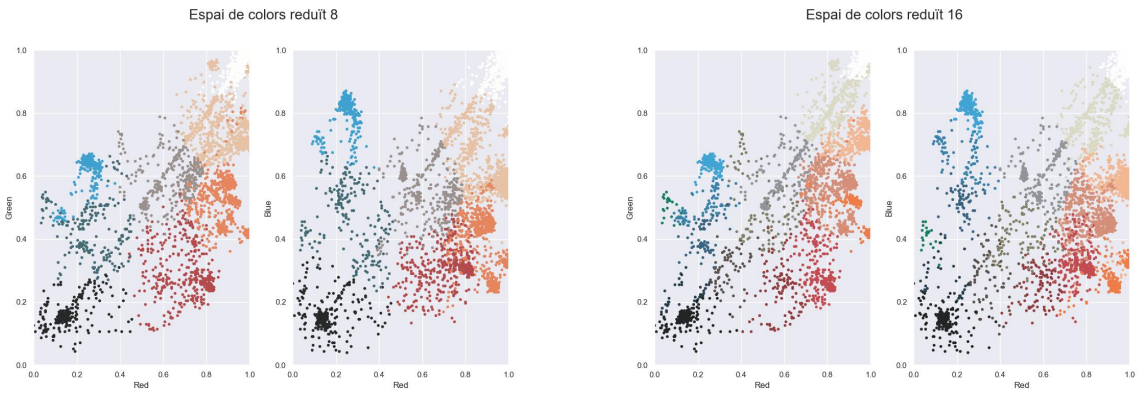

Figura 11: Ordre de K: 8, 16

Codi K-means

```
class KMean:
    def __init__(self,k,max_iter=300):
        self.k = k
        self.centroids = None
        self.max_iter = max_iter

    def initialize_centroids(self, points, k):
        centroids = points.copy()
        np.random.shuffle(centroids)
        centroids = np.unique(centroids, axis=0)
        return centroids[:k]

    def closest_centroid(self, points, centroids):
        distances = np.sqrt(((points - centroids[:, np.newaxis])**2).sum(axis=2))

        return np.argmin(distances, axis=0)
    def move_centroids(self, points, closest, centroids):
        return np.array([points[closest==k].mean(axis=0) for k in range(centroids.
            shape[0])])

    def fit(self, points):
        self.centroids = self.initialize_centroids(points, self.k)
        centroids = None
        for i in range(self.max_iter):
            closest = self.predict(points)
            self.centroids = self.move_centroids(points, closest, self.centroids)
        return closest

    def predict(self, points):
        closest = self.closest_centroid(points, self.centroids)
        return closest
```