ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐHQG HCM
KHOA CÔNG NGHỆ THÔNG TIN

Toán Ứng Dụng Và Thống Kê

Cho Công Nghệ Thông Tin

Báo Cáo

Đồ Án 1: Color Compression

Giảng viên:
Vũ Quốc Hoàng

Trợ giảng:
Phan Thị Phương Uyên
Lê Thanh Tùng
Nguyễn Văn Quang Huy

Sinh viên:

21127657 - Nguyễn Khánh Nhân

TPHCM, tháng 7 năm 2023

Mục lục

I. Thông tin sinh viên

Họ tên: Nguyễn Khánh Nhân
MSSV: 21127657
Lớp: 21CLC02
Email: nknhan21@clc.fitus.edu.vn

II. Thông tin đồ án

1. Giới thiệu vấn đề

- Như đã biết, một bức ảnh hiện nay được lưu trữ dưới dạng một ma trận 2 chiều các điểm ảnh. Tùy thuộc loại ảnh mà các điểm ảnh có cấu tạo khác nhau. Nếu là ảnh xám, mỗi điểm ảnh sẽ được biểu diễn dưới một giá trị không âm, còn nếu là ảnh màu thì mỗi điểm ảnh sẽ được biểu diễn bởi ba giá trị dương tương ứng với mức độ đỏ, xanh dương, xanh lá.
- Chính vì vậy, mỗi bức ảnh sẽ phải chứa hàng triệu màu khác nhau dẫn đến tốn rất nhiều bộ nhớ để lưu chúng. Do đó, vấn đề đặt ra ở đây là chúng ta cần tìm các giảm thiểu số màu có trong một bức ảnh để không tốn quá nhiều dung lượng để lưu, song bên cạnh đó vẫn phải đảm bảo nội dung của bức ảnh.

2. Mục đích đồ án

- Cần giảm số lượng màu trong một bức ảnh xuống theo mong muốn nhưng vẫn đảm bảo được nội dung của bức ảnh.
- Thực hiện gom nhóm các điểm ảnh có màu tương đồng nhau và chọn ra màu đại diện cho từng nhóm, tất cả điểm ảnh thuộc cùng một nhóm sẽ có cùng một màu đại diện.
- Sử dụng thuật toán *"Phân cụm K-means"* thực hiện việc gom nhóm các điểm ảnh có màu tương đồng và tìm ra được màu đại diện cho từng nhóm điểm ảnh giúp đảm bảo nội dung bức ảnh vẫn không bị thay đổi.

III. Ý tưởng thực hiện

1. Thuật toán phân cụm K-means

- Đầu tiên, ta cần tìm hiểu khái quát về thuật toán phân cụm K-means:
    - Phân cụm K-means là một phương pháp ***lượng tử hóa vector***, sử dụng phương pháp tạo và cập nhật trung tâm để phân nhóm các điểm dữ liệu cho trước vào các nhóm khác nhau.
    - Đầu tiên chúng sẽ tạo ra các điểm trung tâm ngẫu nhiên. Sau đó gán mỗi điểm trong tập dữ liệu đầu vào (input) vào một trung tâm gần nó nhất.
    - Cập nhật lại trung tâm mới cho từng cụm dữ liệu, trung tâm mới sẽ là trung bình cộng các điểm dữ liệu trong cụm đó. Lặp lại các bước đã kể trên cho tới khi tất cả các cụm đều không bị thay đổi ở 2 lần lặp kế tiếp nhau ***hoặc*** số lần lặp đã đạt giới hạn cho phép (vì lúc này kết quả đạt được cũng gần đúng và chấp nhận được).

2. Các bước sử dụng K-means trong đồ án

Đầu tiên, ta cần "lượng tử hóa vector" bằng cách xem các điểm ảnh như là một vector. Điều này là hoàn toàn hợp lí vì mỗi điểm ảnh có ba giá trị màu R-G-B và ta có thể xem đây chính là tọa độ của một vector trong không gian 3D.

- Bước 1: Tạo ra các điểm ảnh trung tâm (centroid). Số lượng điểm ảnh trung tâm cũng chính là số màu còn lại của bức ảnh mà ta mong muốn. Có 2 cách thực hiện:
    - Chọn ngẫu nhiên k điểm ảnh từ các điểm ảnh có trong bức ảnh làm điểm ảnh trung tâm ban đầu.
    - Tạo ngẫu nhiên k điểm ảnh có giá trị màu bất kì khác nhau.
- Bước 2: Với mỗi điểm ảnh (vector), ta sẽ tính *độ lệch màu* (còn hiểu là khoảng

cách Euclid giữa hai vector nếu xem điểm ảnh là một vector) của nó tới từng centroid. Gán điểm ảnh đó vào cùng một cụm với centroid có màu gần giống nhất.

- Bước 3: Với mỗi cụm điểm ảnh đã tạo được ở Bước 2, ta sẽ tìm centroid mới cho chúng. Centroid mới có giá trị màu là trung bình cộng giá trị màu của các điểm ảnh trong cùng một cụm. Nếu xem các điểm ảnh là các vector thì centroid mới là vector trung bình của chúng.

- Bước 4: Lặp lại từ bước 2 cho đến khi tất cả các cụm đều không bị thay đổi sau hai lần lặp kế tiếp nhau. Việc đạt được kết quả như vậy là rất khó và tốn thời gian nên thzông thường số lần lặp sẽ được giới hạn tùy ý theo người thiết kế.

IV. Chi tiết về mã nguồn và bố cục của chương trình

1. Thông tin khái quát về chương trình

- Mã nguồn của chương trình được viết hoàn toàn bằng ngôn ngữ lập trình python.
- Các thư viện sử dụng trong mã nguồn: numpy, PIL, matplotlib.
- Trình bày trên một file duy nhất: "21127657.ipynb".

2. Bố cục, nguyên lí hoạt động của chương trình

- Chương trình có tổng cộng 5 hàm phụ và 1 hàm **MAIN**.

• Hàm main(): hàm chính của chương trình.

- Bắt đầu từ hàm **main()**, đầu tiên ta cần cung cấp input trước khi thực hiện thuật toán K-means: tên file ảnh, số lượng màu sau khi nén, số vòng lặp tối đa cho thuật toán K-means (mặc định là 10), cách khởi tạo các centroid (***"in_pixels"***: chọn ngẫu nhiên từ các điểm ảnh, ***"random"***: tạo ra các điểm ảnh có màu sắc

ngẫu nhiên). Việc thực hiện lấy giá trị các input được thực hiện ở hàm **"get_input()"**.

- Trước khi thực hiện thuật toán K-means, ta cần đọc bức ảnh đầu vào nhờ vào hàm **"open"** có trong thư viện **"PIL.Image"**.
- Sau đó thực hiện phân tách bức ảnh thành ma trận các điểm ảnh, vì là ảnh màu nên mỗi điểm ảnh có 3 kênh màu (R-G-B) cho nên ta cần dùng ma trận 3 chiều **"img_arr"** để biểu diễn các điểm ảnh này. Cụ thể là 2 chiều đầu tiên để biểu diễn vị trí của các điểm ảnh và chiều cuối cùng là biểu diễn các kênh màu (channel) của điểm ảnh. Việc chuyển một bức ảnh sang ma trận sẽ được thực hiện nhờ **"numpy.array()"**.
- Để thuận tiện cho việc tính toán, truy xuất các điểm ảnh thì việc dùng ma trận 3 chiều để lưu trữ là không cần thiết, vì vậy ta có thể lưu trữ các điểm ảnh này bằng ma trận 2 chiều **"img_1D_arr"** có kích thước chiều đầu tiên (số dòng) là số lượng điểm ảnh của ảnh gốc và kích thước chiều thứ hai (số cột) là số kênh màu của từng điểm ảnh. Việc thay đổi kích thước ma trận 3 chiều ban đầu thành 2 chiều sẽ được hỗ trợ bởi hàm **"reshape"** có trong thư viện **"numpy"**, tham số truyền vào chính là kích thước của ma trận mới.
- Trước khi thực hiện thuật K-means, ta cần **lưu ý** giá trị màu của các điểm ảnh là từ 0-255, cho nên kiểu dữ liệu của chúng là **"unsigned int 8 bit"** sẽ dẫn đến quá trình tính norm, tìm vector trung bình sẽ bị sai do có thể xuất hiện giá trị thực hoặc nguyên nhưng lớn hơn 255. Vì vậy ta cần phải dùng **astype("float")** để ép kiểu dữ liệu các giá trị màu của điểm ảnh thành **"float"**.
- Sau khi đã có đầy đủ input, ma trận lưu trữ các điểm ảnh thì ta sẽ thực hiện thuật toán K-means nhờ vào hàm **"kmeans()"** (chi tiết về hàm này sẽ được trình bày

bên dưới). Kết quả ta nhận về sẽ là ma trận chứa các centroid **"centroids"** sau khi trải qua việc tính toán và ma trận **"labels"** dùng để thực hiện việc dán nhãn các điểm ảnh, xem chúng sẽ thuộc về nhóm của centroid nào. Ví dụ: **labels[0]** có giá trị là 2 thì đồng nghĩa với sau khi nén màu, trong bức ảnh kết quả, điểm ảnh đầu tiên sẽ có giá trị màu trùng với màu của centroid tại index 2 trong ma trận **"centroids"**.

- **"img_1D_reduced = centroids[labels]"**: sẽ thực hiện thay đổi màu của từng điểm ảnh gốc này thành màu của centroid mà nó thuộc về, sau đó lưu trữ các điểm ảnh mới vào ma trận **"img_1D_reduced"**.
- Để có thể tạo được bức ảnh mới từ ma trận này, ta cần thay đổi kích thước của nó lại như cũ, thành ma trận 3 chiều giống như ma trận **"img_arr"** mà ta có được từ việc đọc ảnh lúc ban đầu, vì vậy kích thước mới của "img_1D_reduced" sẽ giống hệt kích thước của **"img_arr"** (**"img_arr.shape()"**).
- Cuối cùng ta tạo ra bức ảnh mới từ ma trận **"img_1D_reduced"** nhờ vào hàm **"PIL.Image.fromarray()"**. Ta có thể in ra ảnh này trên màn hình nhờ vào hàm **"imshow()"** của thư viện **"matplotlib.pyplot"** cũng như lưu nó dưới 2 định dạng là PNG hoặc PDF (tùy nhu cầu) thông qua hàm **"save()"**.

• Hàm kmeans(): thực hiện thuật toán K-means

- **INPUT:** "img_1d": ma trận chứa các điểm ảnh của ảnh gốc.

  "k_clusters": số cụm sau khi phân vững (số lượng màu sau khi nén).

  "max_iter": số vòng lặp tối đa của thuật toán.

  "init_centroids": cách khởi tạo các centroid ban đầu.

- **OUTPUT:** "centroids": ma trận 2 chiều chứa các centroid và màu của chúng.

"labels": ma trận dùng để đánh dãn các điểm ảnh, xem màu của chúng sẽ thay đổi theo màu của centroid nào ở ảnh mới.

- Dựa vào giá trị của của "init_centroids" mà ta sẽ có cách khởi tạo các centroid khác nhau:
  - "random": Tạo ra ma trận 2 chiều chứa các centroid có giá trị 3 kênh màu ngẫu nhiên nhờ "numpy.random.randint()". Tham số cần có đầu tiên là cận dưới và cận trên của giá trị cần tạo, ở đây là các giá trị kênh màu nên sẽ là (0, 256). Tiếp theo là kích thước ma trận cần tạo, cụ thể là ma trận 2 chiều chứa k centroid, mỗi centroid có 3 kênh màu. Về sau ta sẽ thực hiện tính toán với ma trận này nên cần ép kiểu là "float" để tránh xảy ra lỗi sai số, tràn số,…
  - "in_pixels": Tạo ra các centroid bằng việc chọn ngẫu từ các điểm ảnh. "numpy.random.choice()" sẽ lấy ngẫu nhiên các index của của điểm ảnh trong ma trận "img_1d". Tham số truyền vào là tập các index của điểm ảnh trong ma trận "img_1d", kích thước ma trận "centroids" (= k_clusters) và cuối cùng để đảm bảo không lấy lại các index đã có, ta cần gán giá trị "False" cho tham số "replace". Có được index của những điểm ảnh thì sau đó ta chỉ cần truy xuất chúng ra và gán vào ma trận "centroids". Cần ép kiểu sang "float" như đã nêu ở trên.
- Ta khởi tạo ma trận "labels" là ma trận 1 chiều chứa giá trị "-1" có kích thước bằng tổng số lượng điểm ảnh. Cách khởi tạo đơn giản chỉ là lấy ma trận 1 **"numpy.ones()"** nhân với -1.
- **Thực hiện vòng lặp** để tìm ra các centroid mới và dán nhãn các điểm ảnh sau mỗi lần lặp. Điều kiện dừng là khi số lần lặp đạt giá trị "max_iter" hoặc các giá trị

trong ma trận "labels" không thay đổi sau 2 lần lặp liên tiếp.

- Trước khi thực hiện phân cụm các vector điểm ảnh, ta cần xác định xem màu của điểm ảnh đó gần giống màu của centroid nào nhất hay nếu ta xem điểm ảnh là vector thì vector ấy có khoảng cách Euclid gần centroid nào nhất. Có 2 cách tính:

▪ Cách 1: tự xây dựng hàm tính khoảng cách Euclid cho từng vector điểm ảnh tới tất cả các centroid trong ma trận "centroids". Trong chương trình chính là hàm **"Euclidean_distances(X, Y)"**, với X là ma trận các điểm ảnh, Y là ma trận các centroid. Ta thực hiện 2 vòng lặp vì cần tính khoảng cách từng điểm ảnh trong X với tất cả centroid trong Y. Việc tính toán khoảng cách sẽ được hỗ trợ bởi hàm **"norm()"** có sẵn trong **"numpy.linalg"**. Kết quả trả về là ma trận 2 chiều **"distances"** chứa khoảng cách của từng điểm ảnh tới tất cả các centroid. Ví dụ **"distance[0][2]"** là khoảng cách điểm ảnh thứ 0 đến centroid thứ 2.

▪ Cách 2: vẫn là sử dụng hàm "norm()" như cách 1 nhưng cách sử dụng sẽ hay hơn, không phải sử dụng vòng lặp.

  - Dựa vào kết quả ở cách 1, ta biết kết quả trả về phải là mảng 2 chiều "distances" có kích thước (height x width, k_clusters) với "height", "width" là kích thước ảnh.
  - Vì vậy, dựa vào 2 ma trận có sẵn là **"img_1d"** và **"centroids"**, ta cần tạo ma trận 3 chiều chứa các vector khoảng cách. Với 2 chiều đầu tiên (axis = 0, axis = 1) chứa các vector khoảng cách giữa điểm ảnh và centroid tương ứng, chiều thứ 3 (axis = 2) chính là chiều chứa tọa độ của vector khoảng cách này. Cuối cùng chỉ cần tính khoảng cách Euclid của từng vector khoảng cách.
  - "img_1d" là ma trận có kích thước (height x weight, 3) nên "img_1d[:,

numpy.newaxis]" sẽ là ma trận copy từ "img_1d" nhưng sẽ có thêm một chiều mới nằm giữa 2 chiều cũ của "img_1d". Kích thước của ma trận này là (height x width, 1, 3).

- "img_1d[:, numpy.newaxis] – centroids" sẽ trả về ma trận chứa các vector khoảng cách giữa **từng** điểm ảnh với **tất cả** các centroid. Vì vậy "distance_vectors" sẽ là ma trận 3 chiều có kích thước (height x width, k_clusters, 3).
- Cuối cùng ta sẽ tính độ lớn của các vector khoảng cách bằng **"numpy.linalg.norm()"**. Lưu ý phải gán tham số "axis" là 2 vì chiều axis = 2 chính là chiều chứa các tọa độ của vector khoảng cách, nên việc sử dụng hàm **"norm()"** theo chiều này sẽ giúp tìm được độ lớn của vector khoảng cách. Kết quả cùng là ma trận 2 chiều chứa các khoảng cách giữa điểm ảnh và centroid có kích thước (height x width, k_clusters).
- **Tham số axis**: tham số này xuất hiện trong các hàm tính toán trên ma trận nhiều chiều. Nó sẽ giúp hàm mà ta sử dụng biết được chiều mà nó sẽ thực hiện tính toán trên ma trận. Giá trị mặc định của "axis" là "None", cho biết hàm sử dụng sẽ được thực hiện trên tất cả các chiều của ma trận. Nếu "axis" có giá trị 0 thì hàm sẽ thực hiện tính toán theo cột của ma trận tức là trên các phần tử cùng cột, "axis" có giá trị 1 thì thực hiện tính toán theo dòng tức là trên các phần tử cùng dòng.

- Khi đã có các khoảng cách giữa các vector điểm ảnh và centroid, ta thực hiện việc gán nhãn cho các điểm ảnh. Centroid nào có khoảng cách nhỏ nhất tới điểm ảnh thì sẽ gán nhãn điểm ảnh đó vào centroid ấy nhờ vào hàm **"argmin()"** có trong "numpy". Hàm này sẽ thực hiện tìm vị trí của phần tử bé nhất. Tham số truyền

vào đầu tiên là ma trận chứa các khoảng cách **"distances"**, tiếp theo là tham số "axis" phải bằng **1** vì mục tiêu của ta là tìm khoảng cách nhỏ nhất trong các khoảng cách giữa 1 vector điểm ảnh với **TẤT CẢ** các centroid hay cụ thể là tìm vị trí cột có giá trị nhỏ nhất trên từng hàng của ma trận **"distances"**.

- Sau khi dán nhãn để gom nhóm các điểm ảnh, ta sẽ xem tất cả các nhóm ở lần lặp này có thay đổi gì so với lần lặp trước hay không, nếu không thì đây chính là điều kiện dừng của thuật toán. Việc kiểm tra 2 mảng **"labels"** và **"new_labels"** có giống nhau không được thực hiện nhờ hàm **"numpy.array_equal()"**, nếu có thì giá trị trả về là True rồi dừng thuật toán và ngược lại.
- Việc cuối cùng của mỗi vòng lặp chính là cập nhật lại các centroid mới cho từng cụm. **Có 2 cách** thực hiện việc gom nhóm điểm ảnh trước khi tìm ra centroid mới:

- **Cách 1**: sử dụng hàm tự tạo **"grouping_pixels_1()"** để gom nhóm các điểm ảnh đã được gán nhãn. Xét nhóm thứ i, hàm **"grouping_pixels_1()"** sẽ trả về danh sách (list) các index của các điểm ảnh thuộc nhóm này.
- **Cách 2**: Sử dụng mảng Boolean (Masks) trong numpy: một Masks là một mảng Boolean có cùng kích thước với mảng khác, được sử dụng để chọn hoặc bỏ qua các phần tử của mảng đó. Các phần tử thỏa một điều kiện nào đó thì sẽ được chọn và tại index tương ứng trong Mask sẽ có giá trị **True**, ngược lại là **False**. Như vậy, hàm **"grouping_pixels_2()"** sẽ trả về một mảng Boolean, phần tử thứ i trong mảng này sẽ biểu thị cho việc điểm ảnh thứ i có thuộc về cụm đang xét hay không, nếu có thì phần tử ấy mang giá trị **True**.

- Các điểm ảnh cùng nhóm và giá trị màu của chúng sẽ được lưu trữ trong ma trận 2 chiều **"group_pixels"**. Cuối cùng là tìm vector trung bình của ma trận này bằng hàm **"numpy.mean()"**. Tham số đầu tiên chính là ma trận cần tìm vector trung

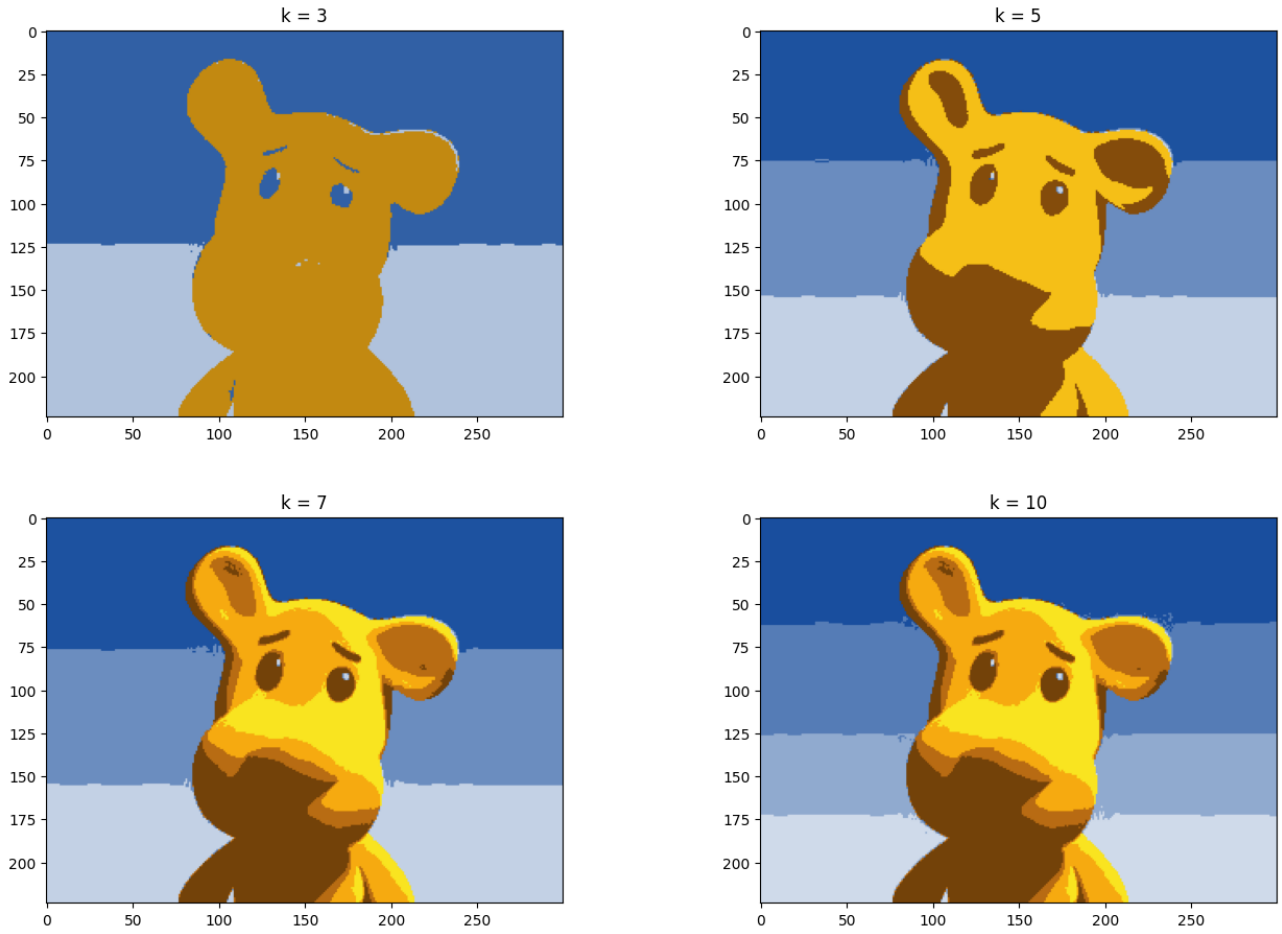
bình **"group_pixels"**. Tham số thứ hai là "axis", vì chiều đầu tiên của ma trận **"group_pixels"** là chiều chứa các vector, chiều thứ 2 là chứa các tọa độ của các vector này nên để tìm vector trung bình của các vector trong ma trận này bằng hàm **"numpy.mean()"** thì giá trị của **axis** phải là 0.

- Trước khi kết thúc vòng lặp hiện tại và sang vòng lặp mới, cần gán các giá trị của **"new_labels"** sang **"labels".**

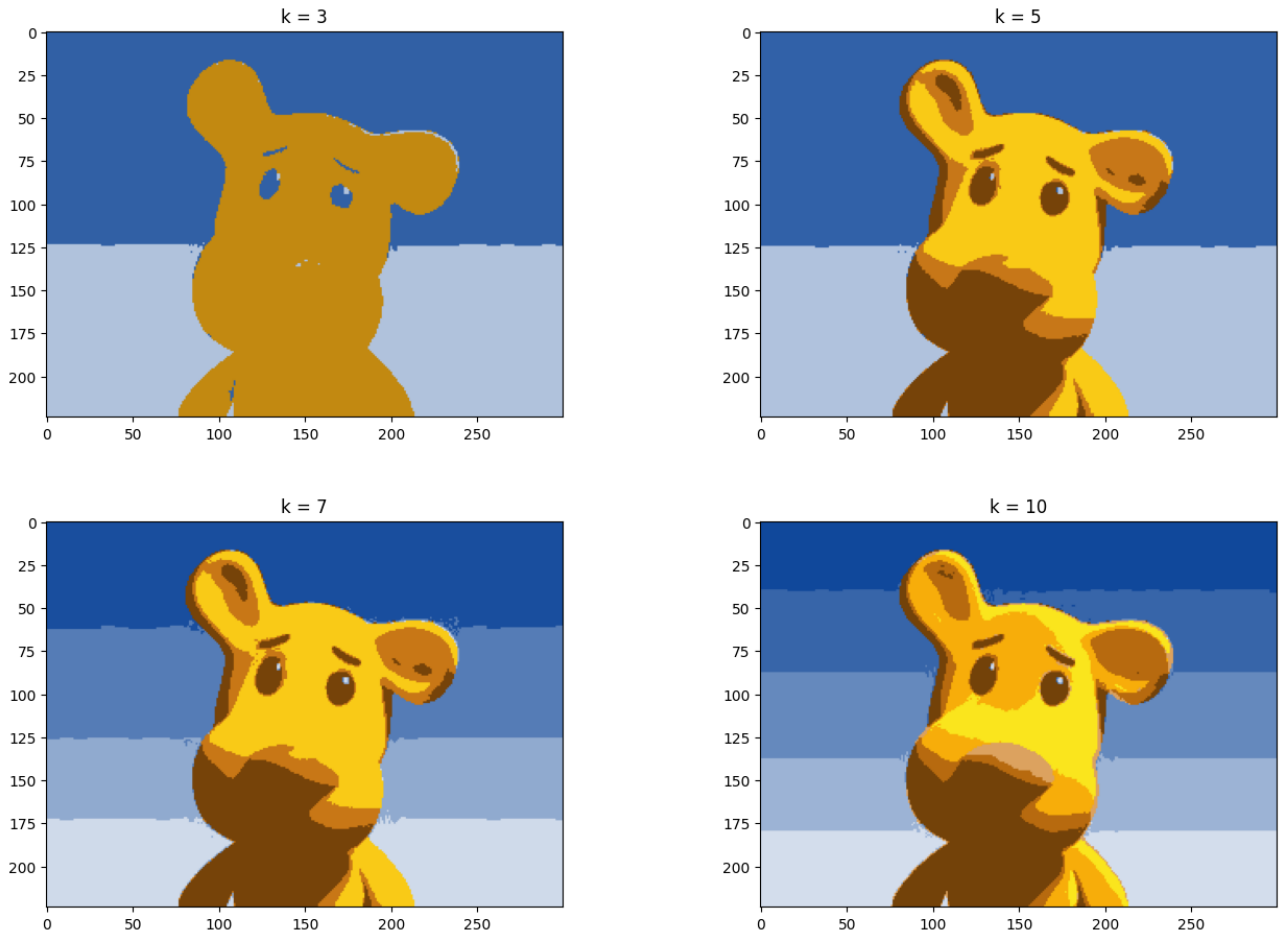
V. Nhận xét, đánh giá kết quả

1. Kết quả và nhận xét

- **Kết quả**



Kết quả ảnh sau khi nén màu với k = 3, 5, 7, 10 ở chế độ "random"



Kết quả ảnh sau khi nén màu với k = 3, 5, 7, 10 ở chế độ "in_pixels"

- **Nhận xét:**

Ảnh đã được nén màu thành công dù ở chế độ "random" ha "in_pixels" chứng tỏ thuật toán "phân cụm K-means" đã hoạt động đúng và tốt. Số lượng màu sau khi nén càng cao thì ảnh càng rỏ nét cũng như sẽ giống ảnh gốc hơn. Dung lượng ảnh cũng sẽ giảm đi nhiều so với ảnh gốc.

Nếu quan sát kĩ thì ta sẽ thấy với cùng giá trị k và số vòng lặp tối đa cho thuật toán thấp thì các ảnh ở chế độ "in_pixels" sẽ có màu rỏ nét và giống ảnh gốc hơn tí sao với "random". Lí do đơn giản là vì cách khởi tạo các centroid ban đầu của chế độ

"in_pixels" là lấy từ ảnh gốc, còn "random" là tạo ra một cách ngẫu nhiên có thể không xuất hiện trong ảnh gốc. Vì vậy nếu số lần lặp ít thì ảnh kết quả ở chế độ "random" có thể xuất hiện màu "lạ" so với ảnh gốc, còn chế độ "in_pixels" chắc chắn không xảy ra hiện tượng này.

2. So sánh, phân tích độ phức tạp

- **Phân tích độ phức tạp của thuật toán K-Means trong chương trình**
  - Trong hàm **kmeans()**, việc khởi tạo ma trận "centroids", "labels" và thực hiện có độ phức tạp về thời gian không đáng kể nên ta sẽ bỏ qua. Ta bắt đầu xét trong từng vòng lặp.
  - Hàm "numpy.linalg.norm()" có độ phức tạp là O(n*k) với n là số lượng điểm ảnh có trong ảnh gốc và k là số centroid (số màu sau khi nén).
  - Hàm "numpy.mean()" có độ phức tạp là O(m) với m là số lượng điểm ảnh trong cùng một nhóm, thực hiện hàm này với k nhóm nên độ phức tạp của việc tìm các centroid mới là O(m*k).
  - Trong trường hợp xấu nhất, thuật toán K-means của chúng ta phải lặp tối đa "max_iter" lần và số lượng điểm ảnh (n) chắc chắn lớn hơn số lượng điểm ảnh trong từng nhóm (m). Vì vậy độ phức tạp cuối cùng của hàm **kmean()** là O[max_iter *n*k] với **"max_iter"** là số lần lặp tối đa của thuật toán, **"n"** là số lượng điểm ảnh (kích thước ảnh) và **"k"** là số màu sau khi nén.
  - Điều này cho thấy nếu bức ảnh có kích thước càng lớn thì việc nén màu ảnh càng mất nhiều thời gian.
- **So sánh việc sử dụng các hàm có sẵn của numpy với các hàm tự tạo**
  - Độ phức tạp về thời gian của hàm **"numpy.linalg.nom()"** và

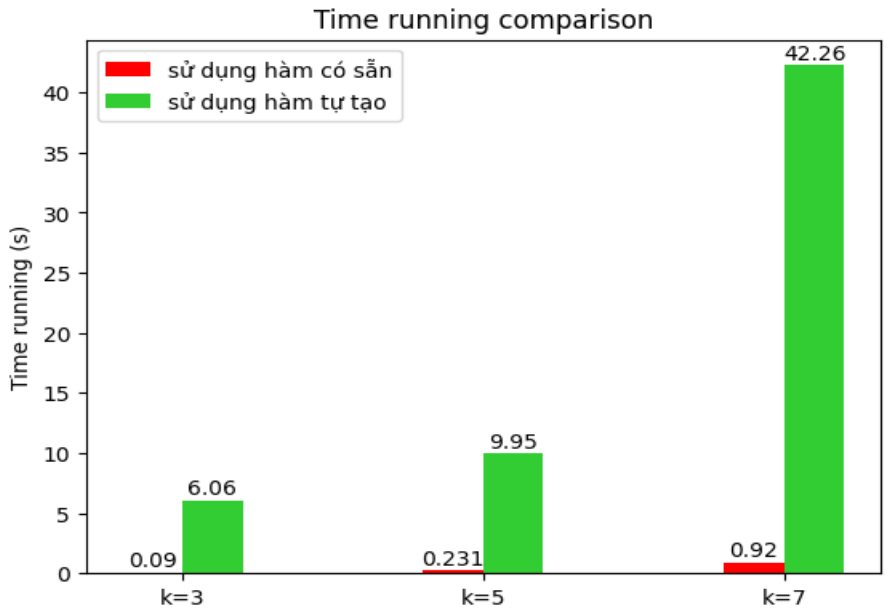
"calc_distances()" là như nhau và bằng **O(n*k)**. Tuy nhiên ta sẽ thấy hàm "numpy.linalg.norm()" sẽ giúp cho thuật toán K-means chạy nhanh hơn rất nhiều so với hàm "calc_distances", *vì sao lại như vậy*?

- Hàm **"numpy.linalg.norm()"** là hàm được cung cấp sẵn nên ta có thể nhận ra là bản chất nó sẽ được viết bằng ngôn ngữ C, ngoài ra nó còn được hỗ trợ bởi các thư viện giúp tối ưu hóa việc tính toán đại số tuyến tính. Điều này giúp nó hoạt động rất hiệu quả và chính xác.
- Hàm **"calc_distances()"** được viết bằng python thuần túy cũng như sử dụng vòng lặp for để tính toán khoảng cách giữa các điểm ảnh và centroid. Chính điều này làm cho nó chạy chậm hơn so với **"numpy.linalg.norm()"** rất nhiều.

- **Chứng minh nhận xét trên**

Kích thước ảnh sử dụng: 300x224

Số vòng lặp tối đa: 100



Biểu đồ cột thể hiện thời gian thực hiện K-means bằng việc sử dụng hàm có sẵn và tự tạo

- **Nhận xét**: Qua biểu đồ trên ta thấy rỏ sự chênh lệch về thời gian thực thi K-means của việc sử dụng các hàm có sẵn trong numpy và việc sử dụng các hàm tự tạo. Đây là minh chứng rỏ ràng nhất cho thấy các hàm được cung cấp sẵn hoạt động với hiệu suất tốn hơn rất nhiều so với các hàm tự viết. Vì vậy nên tìm hiểu và sử dụng các hàm được cung cấp để thuật toán của ta không chỉ nhanh mà còn chính xác.

VI. Tài liệu tham khảo

- Thuật toán K-Means.
- Ứng dụng K-means vào nén màu ảnh
- Cách sử dụng hàm numpy.linalg.norm().
- Cách sử dụng numpy.argmin().
- Cách sử dụng numpy.mean().
- Kĩ thuật sử dụng Masks (mảng boolean) trong numpy.
- Lí do cần phải ép kiểu ma trận điểm ảnh trước khi thực hiện thuật toán K-means.

HẾT