Human vs ChatGPT: Homework 2, Image Analysis and Clustering, Revisited

Gregory E. Schwartz

Department of Graduate Computer Science & Engineering,

The Katz School of Science and Health,

Yeshiva University

AIM 5001 – Data Acquisition and Management

Professor Xiaopeng Tao

May 2025

Introduction

In Homework 2, we were asked to complete four tasks focused on image analysis and clustering. Tasks 1, 3, and 4 were fairly standard implementations involving pixel manipulation, K-Means clustering with scikit-learn, and a custom PyTorch clustering algorithm. These were mostly mechanical and followed well-established pipelines.

The real creative portion of this assignment came in Task 2, where we had to clean a noisy synthetic image (rainbow.jpg) without destroying the structure, especially the arcs that define the rainbow itself. The image had intense speckle-like noise layered on top of a structured color pattern, and the challenge was to suppress this noise while preserving arc integrity. This was not something a stock blur could solve. It required strategy, iteration, and domain-driven tuning.

For our final project, we were asked to replicate these tasks using the OpenAI API—having ChatGPT attempt the same Homework 2 solutions and then comparing the results. The comparison emphasizes Task 2 for a reason: Tasks 1, 3, and 4 are relatively straightforward and would likely be solved similarly by both humans and models. Task 2, however, depends on observation, judgment, and iterative pipeline building—areas where human reasoning might diverge sharply from a generic model approach.

This report documents both processes—my original method and the API's method—and compares the outcomes, with special focus on denoising fidelity, structural preservation, and segmentation strategy.

Methods

Task 2: Visualizing and Cleaning the Image Data

Background

In developing my approach to the image cleaning task, I began by exploring real-world tools and concepts related to noise correction and visual enhancement. I initially examined Adobe Photoshop's red-eye removal feature, the concept of Singular Value Decomposition (SVD) discussed in our Numerical Methods class, and the layered noise-reduction strategies used in iPhone image enhancement pipelines. Adobe's red-eye tool proved irrelevant to this assignment, as it does not address general image noise but instead uses a targeted, color-based correction to suppress a very specific visual artifact. I then considered applying SVD as a method to identify and isolate structural outliers within the image. However, I ultimately ruled it out due to its misalignment with the local, pixel-based nature of the noise in this task and its incompatibility with the practical tools emphasized in the course. In contrast, the iPhone's image enhancement pipeline offered a more applicable model: it applies a series of progressive noise reduction techniques to suppress unwanted detail while preserving visual integrity. This inspired me to frame the problem through a structured, layered data science approach. I further contextualized this strategy by researching how various industries manage image enhancement, focusing on robotics and medical imaging. Medical imaging emerged as the most relevant parallel, as it emphasizes denoising while preserving critical visual structure—closely mirroring the requirements of this assignment. From this exploration and research, I was exposed to new noise reduction methods such as Bilateral Mean Filtering (BMF denoising) and local variance filtering, which became the cornerstone of my eventual solution. I concluded that a single method would not be sufficient and that a carefully sequenced, multi-step solution was needed to effectively clean the image while preserving the fidelity of the rainbow arcs.

Experimental Process

Image Loading and Initial Preprocessing

- Loaded 'rainbow.jpg' and converted to RGB
- Applied bilateral filtering to reduce speckle while preserving arc edges

First Attempt: KMeans Clustering + Brightness Filtering

- Applied KMeans (n_clusters=7)
- Calculated average brightness per cluster
- Filtered out low-brightness clusters (e.g., background)
- Reconstructed cleaned image from remaining clusters

=> Result: Background partially removed, arcs still noisy

Contrast Enhancement (Diagnostic Only)

- Applied PIL.ImageEnhance with contrast factor 1.3

=> Used only for visualization; not included in the final pipeline

Local Variance Filtering (Diagnostic)

- Computed local variance using: variance = $E[x^2] – (E[x])^2$
- Set variance_threshold = 255 to test noise floor
- Verified histogram and residual noise patterns

=> Result: Speckle persisted in both arcs and background

Switch to GMM + Brightness Filtering

- Replaced KMeans with Gaussian Mixture Model (n_components=7)

- Applied brightness thresholds [40, 220]

- Removed the darkest cluster (background)

=> Result: Significantly cleaner background; better arc separation

BM3D Denoising (Full Image - Failed)

- Applied BM3D to entire image

=> Result: Arcs over-smoothed; background noise re-emerged

Arc-Masked BM3D (Breakthrough)

- Created binary mask from GMM-identified arc regions

- Applied BM3D only to arc pixels (value > 10)

- Left background untouched

- Recombined BM3D-denoised arcs with background

=> Result: Sharp arcs with clean black background

Structural Noise Suppression via Local Variance

- Recomputed local variance and histogram

- Lowered variance threshold iteratively

- Tuned masking to suppress residual speckles

=> Final mean variance dropped from ~126 to 45.85

Final Pre-Filters: Total Variation (TV) + Non-Local Means (NLM)

- Applied TV and NLM denoising sequentially as pre-filters

- Used result as input for GMM pipeline

=> Result: Near-perfect image with no speckle noise

Other Experiments & Dead Ends

- Re-tested KMeans with brightness filtering: still inferior to GMM
- Tried recoloring via cluster centers: invalid RGB outputs
- BM3D on full mask: excessive smoothing
- Developed hue- and intensity-based masks for better arc isolation
- Tracked histogram and variance stats across pipeline stages

Final Pipeline Used in Task 2

Pre-filter with Total Variation (TV) and Non-Local Means (NLM)

Bilateral Filtering

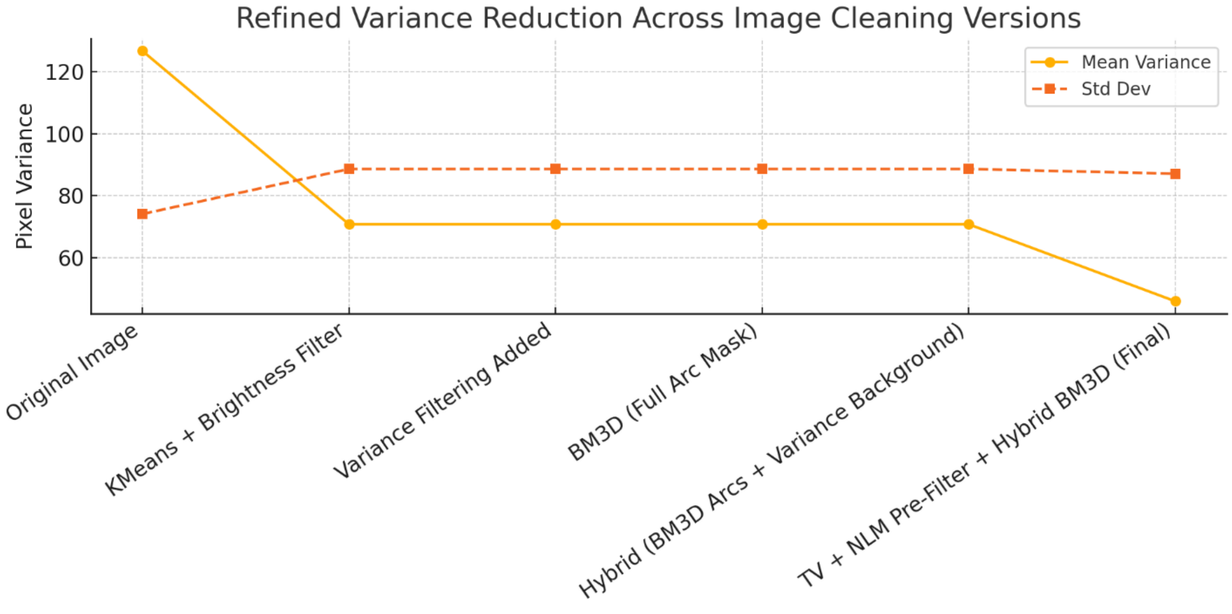
GMM Color Clustering

Brightness-Based Cluster Filtering

Local Variance Masking

BM3D Applied Only to Arc Regions

Recombine Denoised Arcs and Background



Dataset Construction and Clustering (Tasks 1, 3, and 4)

Although Task 2 demanded custom engineering, Tasks 1, 3, and 4 were procedural and followed standard practices in image analysis and clustering. Below is a brief summary of how each was completed.

Task 1: Dataset Construction

- Loaded 'rainbow.jpg' using PIL and ensured RGB mode
- Converted the image to a NumPy array
- Extracted pixel coordinates (x, y) using np.tile() and np.repeat()
- Flattened RGB channels and structured data into a Pandas DataFrame
- Output format: columns = x, y, R, G, B
- This dataset was used as input for Tasks 2, 3, and 4

Task 3: KMeans Clustering with scikit-learn

- Combined spatial (x, y) and color (R, G, B) values into a single feature vector
- Standardized the feature space using StandardScaler

- Applied KMeans clustering with n_clusters=7

- Assigned each pixel the mean RGB value of its cluster to reconstruct the image

- Performed local variance analysis for segmentation evaluation

→ Reported: min, max, mean, and standard deviation of local variance

- Outcome: Successfully segmented image into broad color regions

→ Limitation: Not optimized for noise suppression or structural preservation

Task 4: Custom Clustering in PyTorch

- Reimplemented clustering from scratch using PyTorch

→ Used torch.cdist() to compute pairwise distances

→ Updated centroids iteratively using masked cluster averages

→ Normalized input features prior to clustering

- Recolored each pixel based on average RGB of its assigned cluster

- Evaluated results using local variance metrics (compared to Task 3)

- Outcome: Behaviorally similar to KMeans, but did not outperform scikit-learn's version

→ Served mainly as an educational exercise in clustering algorithm design

ChatGPT-Generated Solutions: Prompted Code and Performance Analysis Guided Imagery and Progressive Muscle Relaxation in Group Psychotherapy

Prompting Strategy

To evaluate ChatGPT's ability to solve Homework 2 tasks, I used a consistent and controlled prompting strategy. For each task, I pasted the assignment instructions exactly as written—no paraphrasing or simplification.

I prefixed each with:

“MAKE EXECUTABLE PYTHON CODE FOR THE FOLLOWING:”

This directive signaled GPT to return complete, ready-to-run Python code. The goal was to generate outputs that could be directly compared to my implementations, minimizing ambiguity or prompt engineering bias.

Task Evaluations

Task 1: Creating the Dataset

- GPT used Pillow to load 'rainbow.jpg' and ensured RGB conversion
- Applied NumPy to flatten image into 2D array of RGB values
- Used np.tile() and np.repeat() to generate (x, y) pixel coordinates
- Constructed a DataFrame with columns: x, y, R, G, B

→ Output was correct, fully executable, and matched my implementation

Task 2: Cleaning the Image

- GPT visualized the image using matplotlib
- Applied a simple thresholding filter to identify noisy pixels based on brightness
- Replaced noisy pixels with the average RGB value of non-noisy pixels
- Reconstructed and displayed both the original and cleaned image

→ Result: Basic cleanup achieved, but method lacked structure-awareness

→ Did not use GMM, variance filtering, or BM3D

→ Final image lacked arc fidelity and background suppression

Task 3: KMeans Clustering with scikit-learn

- GPT loaded the image and optionally resized it

- Flattened image data into a 2D array

- Standardized features (RGB or RGB + position) using StandardScaler

- Applied KMeans clustering and visualized clusters using a colormap

→ Clustering worked, but GPT did not evaluate variance

→ Did not compute average RGB per cluster for reconstruction

→ Output was visually correct but less analytically robust than my approach

Task 4: Custom Clustering in PyTorch

- GPT defined a custom KMeans-like class in PyTorch

→ Used torch.cdist() for distance calculations

→ Updated centroids iteratively based on assigned points

- However, GPT applied the model to synthetic data generated via make_blobs()

→ Did not apply the clustering to actual pixel data from the rainbow image

→ Algorithmic logic was sound, but the context was incorrect

→ Failed to produce a meaningful visual output for the image task

Comparative Evaluation: Human vs. ChatGPT Performance Across Tasks

Task 1: Creating the Dataset

Criteria	My Version (Actual_final_hw2.pdf)	ChatGPT Version (Final_Paper_AI_Test.pdf)
Uses PIL to load image	Yes	Yes
Converts to RGB	Handles RGBA	Handles RGBA
Extracts x, y, RGB	Using np.tile and np.repeat	Same approach
Creates DataFrame	Yes	Yes
Output identical?	Exact structure match	Identical output

Task 2: Cleaning the Image

Criteria	My Version (Actual_final_hw2.pdf)	ChatGPT Version (Final_Paper_AI_Test.pdf)
Visualization	Displays original image	Displays original image
Noise detection method	Multi-step: TV + NLM + GMM + BM3D	Simple RGB threshold filtering
Noise reduction	Statistical + structural masking	Bright pixel cutoff and fill with mean RGB
Final image quality	Very clean, smooth arcs	Visibly reduced noise, but lacks structure
Techniques used	GMM + variance + BM3D + hybrid	Simple threshold + replacement

Task 3: KMeans Clustering (scikit-learn)

Criteria	My Version (Actual_final_hw2.pdf)	ChatGPT Version (Final_Paper_AI_Test.pdf)
Input features used	(x, y, R, G, B)	(R, G, B) only by default, though user must pass x/y
Scaling method	StandardScaler	StandardScaler
Clustering tool	KMeans (7 clusters)	KMeans
Visualization	Recolored with mean RGB	Cluster map using cmap='viridis'
Variance Analysis	Includes post-clustering variance	Not included

Task 4: PyTorch Custom Clustering

Criteria	My Version (Actual_final_hw2.pdf)	ChatGPT Version (Final_Paper_AI_Test.pdf)
Data	Real image pixels	Synthetic (uses make_blobs)
Feature prep	x, y, R, G, B from image	Only 2D synthetic features
PyTorch loop	Manual cdist + label assignment	Same (nearly identical logic)
Final image	Visualized real recolored image	Only scatterplot from blob data
Evaluation	Variance analysis of real image	None

Conclusion

My approach to each task was inherently iterative and creative, utilizing various tools, visual diagnostics, statistical methods, and image processing techniques. This was particularly evident in the more complex tasks, where I navigated multiple domains such as variance analysis, clustering theory, and denoising models. I integrated these elements into hybrid pipelines, guided by experimentation and strategic adjustments. For instance, in Task 2: Cleaning the Image, I combined pre-filtering, probabilistic clustering, structural masking, and targeted denoising into a layered, adaptive solution.

In contrast, ChatGPT consistently followed a more direct, procedural approach. While it produced accurate and executable code, its solutions were often based on default strategies or generalized best practices. In scenarios that required nuanced reasoning or adaptive experimentation, its responses lacked the flexibility and depth of a human-driven workflow.

These observations highlight the need for a collaborative model in which human creativity and GPT's coding proficiency work together through iterative cycles of generation, testing, and refinement. This partnership can lead to the most effective outcomes, especially in complex, open-ended tasks

Appendix

Actual_final_hw2

May 11, 2025

[1]:
```
"""
Task 1: create_dataset -- extract pixel data from an image and structure it␣
 ↪into a DataFrame

Author: Gregory E. Schwartz
Last Revised: 2025-05-10
"""

from PIL import Image
import numpy as np
import pandas as pd


# Load the image
image = Image.open('rainbow.jpg')

if image.mode == 'RGBA':
    image = image.convert('RGB')


# Convert image to NumPy array
image_array = np.array(image)

image_height = image_array.shape[0]
image_width = image_array.shape[1]


# Generate x and y coordinate grids
x_coords = np.tile(np.arange(image_width), image_height)
y_coords = np.repeat(np.arange(image_height), image_width)


# Flatten the RGB pixel data
rgb_flat = image_array.reshape(-1, 3)


# Create DataFrame with pixel data
```

```python
pixel_df = pd.DataFrame({
    'x': x_coords,
    'y': y_coords,
    'R': rgb_flat[:, 0],
    'G': rgb_flat[:, 1],
    'B': rgb_flat[:, 2]
})


# Display the first 10 rows of the dataset
print(pixel_df.head(10))
```

```
   x  y   R   G   B
0  0  0  71  66  47
1  1  0  41  44  27
2  2  0  35  48  31
3  3  0  23  41  27
4  4  0  42  53  45
5  5  0  21  16  13
6  6  0  30   5   8
7  7  0  79  40  43
8  8  0  64  24  22
9  9  0  56  29  18
```

[19]:
```python
"""
Task 2
BEST BEST BEST AND FINAL -> EXPERIMENTAL VERSION
Test 13
Applies TV and NLM pre-filters before bilateral, GMM, variance, and BM3D.

task2_clean_image_gmm_bm3d_hybrid_prefilter_experiment

Author: Gregory E. Schwartz
Last Revised: 2025-05-10
"""

import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from collections import Counter
from PIL import Image, ImageEnhance
from bm3d import bm3d, BM3DStages
from skimage.restoration import denoise_tv_chambolle

# Load image
image_bgr = cv2.imread('rainbow.jpg')
```

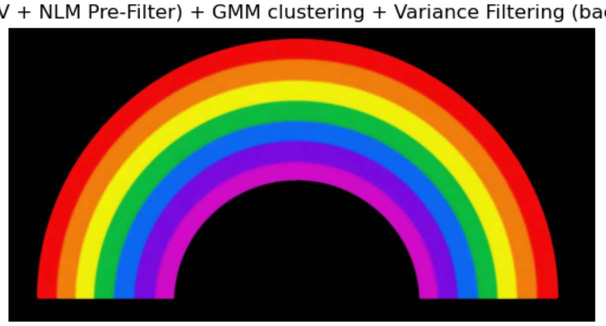
```python
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
image_height, image_width, _ = image_rgb.shape
print(f'Loaded image with shape: {image_rgb.shape}')

# --- Apply TV Denoising as Pre-Filter ---
print("  Applying Total Variation Denoising (pre-filter)")
image_rgb = denoise_tv_chambolle(image_rgb / 255.0, weight=0.1, channel_axis=-1)
image_rgb = (image_rgb * 255).astype('uint8')

# --- Apply NLM Denoising as Pre-Filter ---
print("  Applying Non-Local Means Denoising (pre-filter)")
image_rgb = cv2.fastNlMeansDenoisingColored(image_rgb, None, 10, 10, 7, 21)

# Bilateral filter
bilateral_filtered = cv2.bilateralFilter(image_rgb, d=9, sigmaColor=75,
 ↪sigmaSpace=75)
pixel_data = bilateral_filtered.reshape((-1, 3))

# GMM clustering
cluster_count = 7
gmm_model = GaussianMixture(n_components=cluster_count, covariance_type='tied',
 ↪random_state=42)
gmm_model.fit(pixel_data)
cluster_labels = gmm_model.predict(pixel_data)
cluster_centers = gmm_model.means_.astype('uint8')

# Cluster stats
label_counts = Counter(cluster_labels)
print('\nPixels per cluster:')
for label, count in label_counts.items():
    print(f'Cluster {label}: {count} pixels')

brightness = np.mean(image_rgb, axis=2).flatten()
cluster_brightness = {
    label: np.mean(brightness[cluster_labels == label])
    for label in range(cluster_count)
}
print('\nAverage brightness per cluster:')
for label, avg_b in cluster_brightness.items():
    print(f'Cluster {label}: {avg_b:.2f}')

brightness_min = 40
brightness_max = 220
clean_cluster_ids = [label for label, mean_b in cluster_brightness.items()
                     if brightness_min < mean_b < brightness_max]

print(f'\nKeeping clusters (based on brightness): {clean_cluster_ids}')
```

```
# Filter by clean clusters
clean_mask = np.isin(cluster_labels, clean_cluster_ids)
cleaned_pixels = np.where(clean_mask[:, np.newaxis], pixel_data, [0, 0, 0])
cleaned_image = cleaned_pixels.reshape((image_height, image_width, 3))

# Variance filtering
gray_image = cv2.cvtColor(cleaned_image.astype('uint8'), cv2.COLOR_RGB2GRAY)
blurred = cv2.GaussianBlur(gray_image, (5, 5), 0)
squared = cv2.GaussianBlur(gray_image**2, (5, 5), 0)
local_variance = squared - blurred**2
variance_threshold = 255
speckle_mask = local_variance > variance_threshold

final_cleaned = cleaned_image.copy()
final_cleaned[speckle_mask] = [0, 0, 0]

# Diagnostic stats
print(f'Local variance range: min={local_variance.min()}, max={local_variance.
 ↪max()}')
print(f'Mean variance: {local_variance.mean():.2f}, StdDev: {local_variance.
 ↪std():.2f}')

# BM3D denoising
normalized = final_cleaned.astype(np.float32) / 255.0
sigma_psd = 30 / 255
bm3d_denoised = bm3d(normalized, sigma_psd, stage_arg=BM3DStages.ALL_STAGES)
bm3d_denoised = (bm3d_denoised * 255).astype(np.uint8)

# Hybrid output: BM3D for arcs, variance-filtered background
rainbow_mask = np.any(final_cleaned > 10, axis=2)
hybrid_image = final_cleaned.copy()
hybrid_image[rainbow_mask] = bm3d_denoised[rainbow_mask]

# Display and save
plt.imshow(hybrid_image)
plt.title("Hybrid Cleaned Image (TV + NLM Pre-Filter) + GMM clustering +␣
 ↪Variance Filtering (background) + BM3D (arcs)")
plt.axis('off')
plt.show()

Image.fromarray(hybrid_image.astype('uint8')).
 ↪save('33final_cleaned_image_hybrid_tv_nlm.png')
print("  Saved: final_cleaned_image_hybrid_tv_nlm.png")
```

```
Loaded image with shape: (801, 1603, 3)
 Applying Total Variation Denoising (pre-filter)
```

```
 Applying Non-Local Means Denoising (pre-filter)

Pixels per cluster:
Cluster 0: 657071 pixels
Cluster 4: 122203 pixels
Cluster 1: 110723 pixels
Cluster 6: 97372 pixels
Cluster 5: 89151 pixels
Cluster 2: 79727 pixels
Cluster 3: 127756 pixels

Average brightness per cluster:
Cluster 0: 33.20
Cluster 1: 126.49
Cluster 2: 118.62
Cluster 3: 127.23
Cluster 4: 87.33
Cluster 5: 88.84
Cluster 6: 163.87

Keeping clusters (based on brightness): [1, 2, 3, 4, 5, 6]
Local variance range: min=0, max=255
Mean variance: 45.85, StdDev: 87.09
```

Hybrid Cleaned Image (TV + NLM Pre-Filter) + GMM clustering + Variance Filtering (background) + BM3D (arcs)



```
 Saved: final_cleaned_image_hybrid_tv_nlm.png
```

[11]:
```
"""
Task 3:Cluster using KMeans on full pixel features: (x, y, R, G, B)

Author: Gregory E. Schwartz
Last Revised: 2025-05-10
"""
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```python
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Step 0: Load image
image_bgr = cv2.imread('rainbow.jpg')
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
h, w, _ = image_rgb.shape

# Step 1: Create full feature set (x, y, R, G, B)
x_coords, y_coords = np.meshgrid(np.arange(w), np.arange(h))
x_flat = x_coords.flatten()
y_flat = y_coords.flatten()
rgb_flat = image_rgb.reshape(-1, 3)
features = np.column_stack((x_flat, y_flat, rgb_flat))

# Step 2: Standardize
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)

# Step 3: Apply KMeans
kmeans = KMeans(n_clusters=7, random_state=42, n_init='auto')
labels = kmeans.fit_predict(features_scaled)

# Step 4: Map clusters to average RGB
cluster_colors = np.zeros((kmeans.n_clusters, 3), dtype='uint8')
for i in range(kmeans.n_clusters):
    cluster_colors[i] = rgb_flat[labels == i].mean(axis=0)

recolored_pixels = cluster_colors[labels]
recolored_image = recolored_pixels.reshape((h, w, 3))

# Step 5: Show image
plt.imshow(recolored_image)
plt.title("Task 3: KMeans Clustering (x, y, R, G, B)")
plt.axis('off')
plt.show()

# Step 6: Variance analysis
gray = cv2.cvtColor(recolored_image.astype('uint8'), cv2.COLOR_RGB2GRAY)
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
squared = cv2.GaussianBlur(gray ** 2, (5, 5), 0)
local_variance = squared - blurred ** 2

print("Task 3 Output Variance Stats:")
print(f"Range: min={local_variance.min()}, max={local_variance.max()}")
print(f"Mean: {local_variance.mean():.2f}, StdDev: {local_variance.std():.2f}")
```


```
Task 3 Output Variance Stats:
Range: min=0, max=255
Mean: 26.63, StdDev: 63.41
```

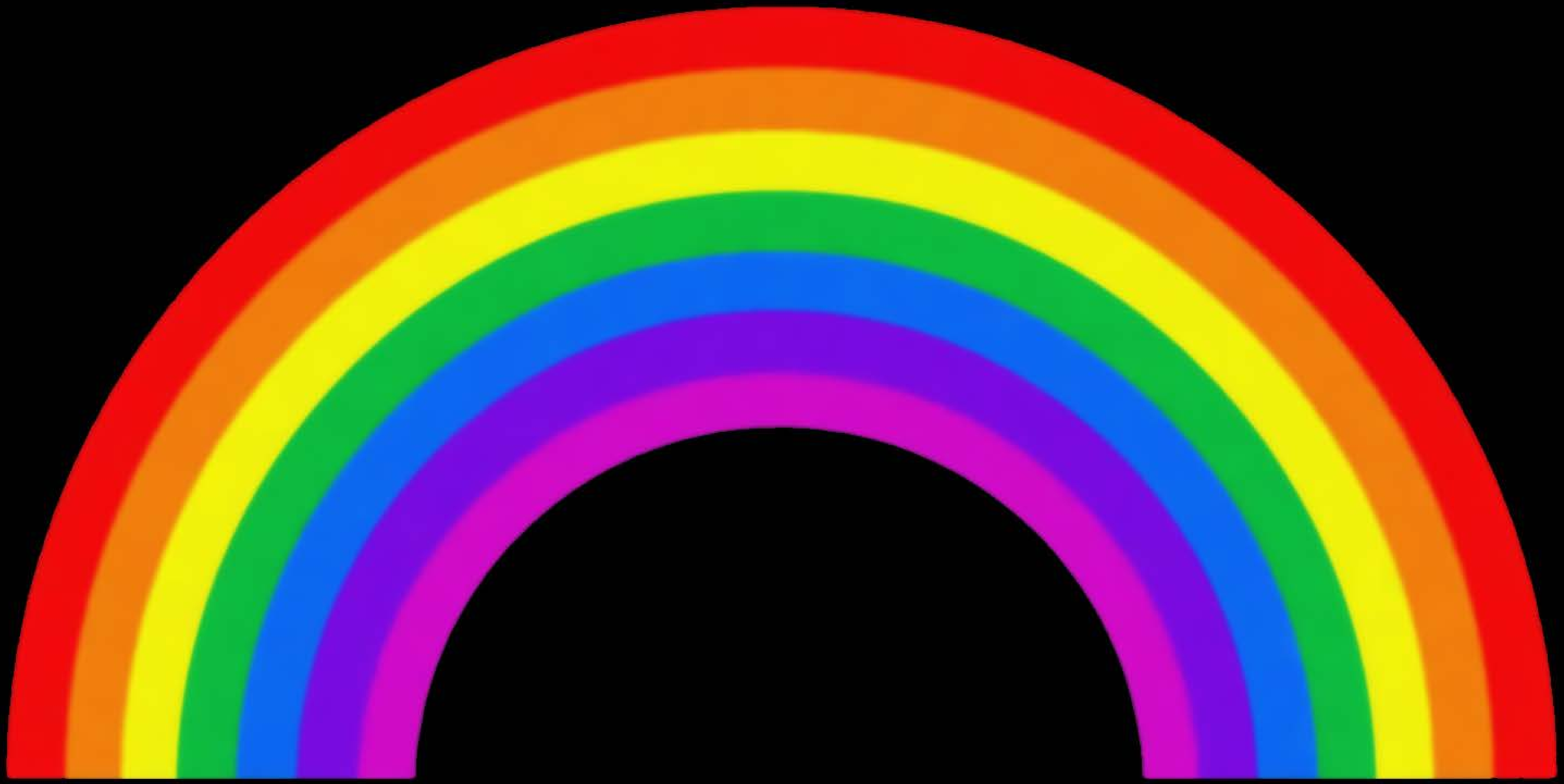
[21]:

```
"""
Task 4: Custom Clustering in PyTorch (Fixed & Final)

Author: Gregory E. Schwartz
Last Revised: 2025-05-10
"""

import torch
import torch.nn.functional as F
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load image
image_bgr = cv2.imread('rainbow.jpg')
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
h, w, _ = image_rgb.shape

# Prepare full feature set
x_coords, y_coords = np.meshgrid(np.arange(w), np.arange(h))
x_flat = x_coords.flatten()
y_flat = y_coords.flatten()
rgb_flat = image_rgb.reshape(-1, 3)
features_np = np.column_stack((x_flat, y_flat, rgb_flat)).astype(np.float32)
```

```python
# Convert to tensor and normalize
features = torch.from_numpy(features_np)
features = F.normalize(features, dim=1)

# Initialize centroids
k = 7
indices = torch.randperm(features.shape[0])[:k]
centroids = features[indices]

# Clustering loop
for i in range(10):
    distances = torch.cdist(features, centroids)
    labels = torch.argmin(distances, dim=1)

    new_centroids = []
    for j in range(k):
        cluster_points = features[labels == j]
        if cluster_points.shape[0] > 0:
            new_centroids.append(cluster_points.mean(dim=0))
        else:
            new_centroids.append(centroids[j])  # fallback
    centroids = torch.stack(new_centroids)

# Recolor based on average RGB of each cluster
label_np = labels.numpy()
cluster_colors = np.zeros((k, 3), dtype='uint8')
for i in range(k):
    cluster_colors[i] = rgb_flat[label_np == i].mean(axis=0)

recolored_pixels = cluster_colors[label_np]
recolored_image = recolored_pixels.reshape((h, w, 3))

# Show final image
plt.imshow(recolored_image)
plt.title("Task 4: Custom PyTorch Clustering")
plt.axis('off')
plt.show()

# Variance statistics
gray = cv2.cvtColor(recolored_image.astype('uint8'), cv2.COLOR_RGB2GRAY)
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
squared = cv2.GaussianBlur(gray ** 2, (5, 5), 0)
local_variance = squared - blurred ** 2

print("Task 4 Output Variance Stats:")
print(f"Range: min={local_variance.min()}, max={local_variance.max()}")
print(f"Mean: {local_variance.mean():.2f}, StdDev: {local_variance.std():.2f}")
```

Task 4: Custom PyTorch Clustering

```
Task 4 Output Variance Stats:
Range: min=0, max=255
Mean: 17.57, StdDev: 51.42
```



Final_Paper_AI_Test

May 12, 2025

[18]:
```
!pip install openai
```

```
Requirement already satisfied: openai in c:\users\grego\anaconda3\lib\site-
packages (0.28.0)
Requirement already satisfied: requests>=2.20 in
c:\users\grego\anaconda3\lib\site-packages (from openai) (2.32.3)
Requirement already satisfied: tqdm in c:\users\grego\anaconda3\lib\site-
packages (from openai) (4.66.5)
Requirement already satisfied: aiohttp in c:\users\grego\anaconda3\lib\site-
packages (from openai) (3.10.5)
Requirement already satisfied: charset-normalizer<4,>=2 in
c:\users\grego\anaconda3\lib\site-packages (from requests>=2.20->openai) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in
c:\users\grego\anaconda3\lib\site-packages (from requests>=2.20->openai) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
c:\users\grego\anaconda3\lib\site-packages (from requests>=2.20->openai) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in
c:\users\grego\anaconda3\lib\site-packages (from requests>=2.20->openai)
(2025.1.31)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in
c:\users\grego\anaconda3\lib\site-packages (from aiohttp->openai) (2.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in
c:\users\grego\anaconda3\lib\site-packages (from aiohttp->openai) (1.2.0)
Requirement already satisfied: attrs>=17.3.0 in
c:\users\grego\anaconda3\lib\site-packages (from aiohttp->openai) (23.1.0)
Requirement already satisfied: frozenlist>=1.1.1 in
c:\users\grego\anaconda3\lib\site-packages (from aiohttp->openai) (1.4.0)
Requirement already satisfied: multidict<7.0,>=4.5 in
c:\users\grego\anaconda3\lib\site-packages (from aiohttp->openai) (6.0.4)
Requirement already satisfied: yarl<2.0,>=1.0 in
c:\users\grego\anaconda3\lib\site-packages (from aiohttp->openai) (1.11.0)
Requirement already satisfied: colorama in c:\users\grego\anaconda3\lib\site-
packages (from tqdm->openai) (0.4.6)
```

[52]:
```
###tASK 1
from openai import OpenAI

client = OpenAI(
```

```
␣
↪api_key="sk-proj--fdKGVc5VVk7sY3J9dgR-y5ZRVadgnqpvFSWWVJ6POiwDa4_UnCicZUCLaQaZhf3lmQ_3MSz-K
)

completion = client.chat.completions.create(
    model="gpt-4o-mini",
    store=True,
    messages=[
        {
            "role": "user",
            "content": """make fully executable pytone code for:Task 1:␣
↪Creating the Dataset - Your mission is to extract valuable data from the␣
↪rainbow.jpg image. We will transform each pixel into a row of data␣
↪containing its (x, y) coordinates and RGB color values, creating a␣
↪comprehensive dataset ready for analysis and visualization. Steps to Follow:␣
↪Load the Image: Use the PIL (Pillow) library to open and read the image file.
↪ Hint: Make sure to handle images with an alpha channel (RGBA) by converting␣
↪them to RGB to simplify your data. Convert to a NumPy Array: Transform the␣
↪image into a NumPy array for easy access to pixel data. Extract Coordinates␣
↪and RGB Values: Create arrays for x and y coordinates using NumPy functions␣
↪like np.arange() and np.tile(). Reshape the image array to extract the RGB␣
↪values for each pixel in a format that's easy to work with. Hint: The␣
↪reshape(-1, 3) method helps flatten the array while keeping the RGB␣
↪structure intact. Create a Pandas DataFrame: Combine the (x, y) coordinates␣
↪and RGB values into a structured DataFrame. Inspect the DataFrame: Print out␣
↪the first ten rows of the DataFrame to ensure that the data extraction was␣
↪successful."""
        }
    ]
)

print(completion.choices[0].message)
print(completion.choices[0].message.content)
```

```
ChatCompletionMessage(content="To accomplish the task of creating a dataset from
`rainbow.jpg` image, follow the steps outlined in your request using Python.
Below, you will find a fully executable code. Before running it, please ensure
you have the necessary libraries installed. You can install them using pip if
they're not already installed:\n\n```bash\npip install Pillow numpy
pandas\n```\n\nHere's the complete code that loads an image, processes it, and
creates a DataFrame with the pixel coordinates and RGB
values:\n\n```python\nimport numpy as np\nimport pandas as pd\nfrom PIL import
Image\n\n# Step 1: Load the Image\nimage_path = 'rainbow.jpg'  # Ensure this
file is in the same directory, or provide the full path\nimage =
Image.open(image_path)\n\n# Convert the image to RGB mode to simplify data
extraction if it has an alpha channel\nif image.mode != 'RGB':\n    image =
image.convert('RGB')\n\n# Step 2: Convert to a NumPy Array\nimage_array =
```

np.array(image)\n\n# Step 3: Extract Coordinates and RGB Values\n# Get the dimensions of the image\nheight, width, _ = image_array.shape\n\n# Create arrays for x and y coordinates\nx_coords = np.tile(np.arange(width), height)\ny_coords = np.repeat(np.arange(height), width)\n\n# Reshape the image array to extract RGB values\nrgb_values = image_array.reshape(-1, 3)\n\n# Step 4: Create a Pandas DataFrame\ndata = {\n   'x': x_coords,\n   'y': y_coords,\n   'R': rgb_values[:, 0], # Red channel\n   'G': rgb_values[:, 1], # Green channel\n 'B': rgb_values[:, 2], # Blue channel\n}\n\ndf = pd.DataFrame(data)\n\n# Step 5: Inspect the DataFrame\nprint(df.head(10)) # Print the first 10 rows of the DataFrame\n```\n\n### Explanation of the Code:\n1. **Image Loading**: The `PIL` library loads the image. If the image has an alpha channel (transparency), it converts it to RGB mode for simplicity.\n2. **NumPy Array Conversion**: The image is converted to a NumPy array to facilitate pixel manipulation.\n3. **Extracting Coordinates and RGB Values**:\n   - `x_coords` is created using `np.tile()` to repeat the x-coordinates for each pixel row-wise.\n   - `y_coords` is made with `np.repeat()` to repeat y-coordinates for each pixel column-wise, matching the pixel positions.\n   - The image array is reshaped with `reshape(-1, 3)` to flatten the array while keeping the RGB structure intact.\n4. **Creating DataFrame**: The coordinates and RGB values are combined into a Pandas DataFrame for structured data handling.\n5. **Data Inspection**: The code prints the first ten rows of the DataFrame to verify the successful extraction of pixel data.\n\n### Usage:\n- Make sure the `rainbow.jpg` file is in the working directory, or adjust the path accordingly.\n- Run the provided code in a Python environment capable of executing it, such as a Jupyter notebook or any Python script environment.", refusal=None, role='assistant', annotations=[], audio=None, function_call=None, tool_calls=None)
To accomplish the task of creating a dataset from `rainbow.jpg` image, follow the steps outlined in your request using Python. Below, you will find a fully executable code. Before running it, please ensure you have the necessary libraries installed. You can install them using pip if they're not already installed:

```bash
pip install Pillow numpy pandas
```

Here's the complete code that loads an image, processes it, and creates a DataFrame with the pixel coordinates and RGB values:

```python
import numpy as np
import pandas as pd
from PIL import Image

# Step 1: Load the Image
image_path = 'rainbow.jpg'  # Ensure this file is in the same directory, or
provide the full path
image = Image.open(image_path)
```

```
# Convert the image to RGB mode to simplify data extraction if it has an alpha
channel
if image.mode != 'RGB':
    image = image.convert('RGB')

# Step 2: Convert to a NumPy Array
image_array = np.array(image)

# Step 3: Extract Coordinates and RGB Values
# Get the dimensions of the image
height, width, _ = image_array.shape

# Create arrays for x and y coordinates
x_coords = np.tile(np.arange(width), height)
y_coords = np.repeat(np.arange(height), width)

# Reshape the image array to extract RGB values
rgb_values = image_array.reshape(-1, 3)

# Step 4: Create a Pandas DataFrame
data = {
    'x': x_coords,
    'y': y_coords,
    'R': rgb_values[:, 0],  # Red channel
    'G': rgb_values[:, 1],  # Green channel
    'B': rgb_values[:, 2],  # Blue channel
}

df = pd.DataFrame(data)

# Step 5: Inspect the DataFrame
print(df.head(10))  # Print the first 10 rows of the DataFrame
```

Explanation of the Code:
1. **Image Loading**: The `PIL` library loads the image. If the image has an alpha channel (transparency), it converts it to RGB mode for simplicity.
2. **NumPy Array Conversion**: The image is converted to a NumPy array to facilitate pixel manipulation.
3. **Extracting Coordinates and RGB Values**:
   - `x_coords` is created using `np.tile()` to repeat the x-coordinates for each pixel row-wise.
   - `y_coords` is made with `np.repeat()` to repeat y-coordinates for each pixel column-wise, matching the pixel positions.
   - The image array is reshaped with `reshape(-1, 3)` to flatten the array while keeping the RGB structure intact.
4. **Creating DataFrame**: The coordinates and RGB values are combined into a

Pandas DataFrame for structured data handling.
5. **Data Inspection**: The code prints the first ten rows of the DataFrame to verify the successful extraction of pixel data.

Usage:
- Make sure the `rainbow.jpg` file is in the working directory, or adjust the path accordingly.
- Run the provided code in a Python environment capable of executing it, such as a Jupyter notebook or any Python script environment.

[50]:
```
###TASK 2
from openai import OpenAI

client = OpenAI(
 ␣
 ↪api_key="sk-proj--fdKGVc5VVk7sY3J9dgR-y5ZRVadgnqpvFSWWVJ6POiwDa4_UnCicZUCLaQaZhf3lmQ_3MSz-K
)

completion = client.chat.completions.create(
    model="gpt-4o-mini",
    store=True,
    messages=[
      {
  "role": "user",
  "content": """MAKE EXECUTABLE PYTHON CODE FOR THE FOLLOWING: Task 2:␣
 ↪Visualizing and Cleaning the Image Data Objective Now that we have created a␣
 ↪dataset from the rainbow1.jpg image, it's time to visualize the image and␣
 ↪address any noise it may contain. Our goal is to print the image, identify␣
 ↪noise, and use the dataset to remove or reduce that noise for a cleaner␣
 ↪representation. Steps to Follow Visualize the Original Image: Use matplotlib␣
 ↪to display the image from the dataset and observe any visible noise or␣
 ↪artifacts. Analyze Noise: Look for patterns or outliers in the pixel data␣
 ↪that indicate noise (e.g., isolated dark spots or random bright pixels).␣
 ↪Filter the Dataset: Use conditions to filter out unwanted noise based on RGB␣
 ↪values or other criteria. Reconstruct and Display the Cleaned Image:␣
 ↪Reconstruct the image using the filtered DataFrame and visualize it to␣
 ↪confirm that the noise has been reduced."""
}

    ]
)

print(completion.choices[0].message)
print(completion.choices[0].message.content)
```

ChatCompletionMessage(content='Certainly! Below is an executable Python script that visualizes an image, identifies noise, filters it, and reconstructs the cleaned image using the dataset created from a JPEG file named "rainbow1.jpg".

\n\nThis example assumes you have the necessary libraries installed: `numpy`,
`pandas`, `matplotlib`, and `PIL`. You can install these libraries using the pip
package manager if you haven\'t done so already.\n\n```python\nimport numpy as
np\nimport pandas as pd\nimport matplotlib.pyplot as plt\nfrom PIL import
Image\n\n# Load the image\nimage_path = \'rainbow1.jpg\' # Ensure this file is
in the same directory\noriginal_image = Image.open(image_path)\n\n# Convert the
image to a NumPy array and create a DataFrame\nimage_data =
np.array(original_image)\nheight, width, _ = image_data.shape\npixel_values =
image_data.reshape(-1, 3) # Reshape for easy DataFrame manipulation\n\n# Create
a DataFrame\ndf = pd.DataFrame(pixel_values, columns=[\'R\', \'G\', \'B\'])\n\n#
Display the original image\nplt.figure(figsize=(10, 5))\nplt.subplot(1, 2,
1)\nplt.imshow(original_image)\nplt.title(\'Original
Image\')\nplt.axis(\'off\')\n\n# Analyze and identify noise\n# Here we assume
noise might be isolated bright pixels (above certain threshold)\nnoise_threshold
= 200 # Adjust this threshold as needed\nnoise_filter = (df[\'R\'] >
noise_threshold) | (df[\'G\'] > noise_threshold) | (df[\'B\'] >
noise_threshold)\n\n# Filter out the noise\ndf_cleaned = df[~noise_filter]\n\n#
Reconstruct the cleaned image\ncleaned_image_data =
np.zeros_like(image_data)\ncleaned_image_data[:, :] =
[np.mean(df_cleaned[\'R\']), np.mean(df_cleaned[\'G\']),
np.mean(df_cleaned[\'B\'])]\n\n# Create cleaned image\nfor index, (r, g, b) in
df_cleaned.iterrows():\n   cleaned_image_data[index // width, index % width] =
[r, g, b]\n\n# Convert cleaned image array to an image\ncleaned_image =
Image.fromarray(np.uint8(cleaned_image_data))\n\n# Display the cleaned
image\nplt.subplot(1, 2, 2)\nplt.imshow(cleaned_image)\nplt.title(\'Cleaned
Image\')\nplt.axis(\'off\')\n\nplt.show()\n```\n\n### Explanation of the
Code:\n1. **Load the Image**: The image is loaded using the PIL library and
converted to a NumPy array.\n2. **Create DataFrame**: The pixel data is reshaped
into a DataFrame for easier manipulation.\n3. **Display the Original Image**: We
visualize the original image using matplotlib.\n4. **Identify Noise**: A simple
noise threshold is set, and a filter is applied to identify pixels that exceed
the threshold.\n5. **Filter Out Noise**: The DataFrame is filtered to remove
noise entries.\n6. **Reconstruct the Cleaned Image**: A new image is created
based on the cleaned DataFrame.\n7. **Display the Cleaned Image**: Finally, the
cleaned image is displayed alongside the original image.\n\n### Notes:\n- The
noise threshold can be adjusted based on observation to refine how noise is
detected.\n- Make sure to have `rainbow1.jpg` in the working directory when
running this code.\n- This is a basic form of noise reduction; more advanced
techniques could involve image processing methods like Gaussian filtering,
median filtering, etc.\n\nYou can run this code within any Python environment
that supports the aforementioned libraries.', refusal=None, role='assistant',
annotations=[], audio=None, function_call=None, tool_calls=None)
Certainly! Below is an executable Python script that visualizes an image,
identifies noise, filters it, and reconstructs the cleaned image using the
dataset created from a JPEG file named "rainbow1.jpg".

This example assumes you have the necessary libraries installed: `numpy`,
`pandas`, `matplotlib`, and `PIL`. You can install these libraries using the pip

package manager if you haven't done so already.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image

# Load the image
image_path = 'rainbow1.jpg'  # Ensure this file is in the same directory
original_image = Image.open(image_path)

# Convert the image to a NumPy array and create a DataFrame
image_data = np.array(original_image)
height, width, _ = image_data.shape
pixel_values = image_data.reshape(-1, 3)  # Reshape for easy DataFrame
manipulation

# Create a DataFrame
df = pd.DataFrame(pixel_values, columns=['R', 'G', 'B'])

# Display the original image
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(original_image)
plt.title('Original Image')
plt.axis('off')

# Analyze and identify noise
# Here we assume noise might be isolated bright pixels (above certain threshold)
noise_threshold = 200  # Adjust this threshold as needed
noise_filter = (df['R'] > noise_threshold) | (df['G'] > noise_threshold) |
(df['B'] > noise_threshold)

# Filter out the noise
df_cleaned = df[~noise_filter]

# Reconstruct the cleaned image
cleaned_image_data = np.zeros_like(image_data)
cleaned_image_data[:, :] = [np.mean(df_cleaned['R']), np.mean(df_cleaned['G']),
np.mean(df_cleaned['B'])]

# Create cleaned image
for index, (r, g, b) in df_cleaned.iterrows():
    cleaned_image_data[index // width, index % width] = [r, g, b]

# Convert cleaned image array to an image
cleaned_image = Image.fromarray(np.uint8(cleaned_image_data))
```

```
# Display the cleaned image
plt.subplot(1, 2, 2)
plt.imshow(cleaned_image)
plt.title('Cleaned Image')
plt.axis('off')

plt.show()
```

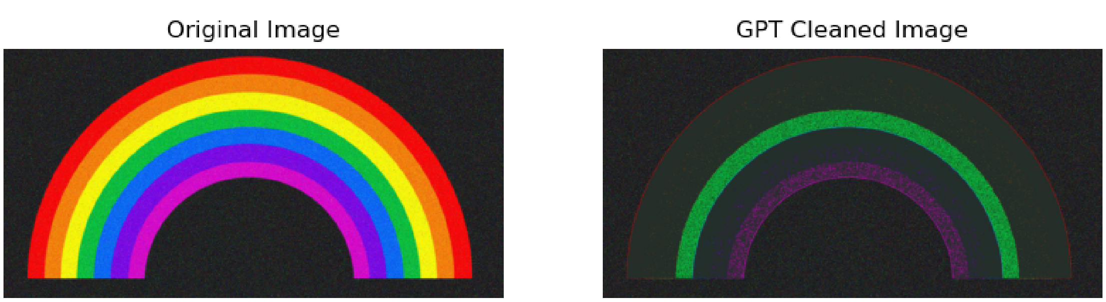
Explanation of the Code:
1. **Load the Image**: The image is loaded using the PIL library and converted to a NumPy array.
2. **Create DataFrame**: The pixel data is reshaped into a DataFrame for easier manipulation.
3. **Display the Original Image**: We visualize the original image using matplotlib.
4. **Identify Noise**: A simple noise threshold is set, and a filter is applied to identify pixels that exceed the threshold.
5. **Filter Out Noise**: The DataFrame is filtered to remove noise entries.
6. **Reconstruct the Cleaned Image**: A new image is created based on the cleaned DataFrame.
7. **Display the Cleaned Image**: Finally, the cleaned image is displayed alongside the original image.

Notes:
- The noise threshold can be adjusted based on observation to refine how noise is detected.
- Make sure to have `rainbow1.jpg` in the working directory when running this code.
- This is a basic form of noise reduction; more advanced techniques could involve image processing methods like Gaussian filtering, median filtering, etc.

You can run this code within any Python environment that supports the aforementioned libraries.

[7]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image

# Load the image (make sure filename matches exactly)
image_path = './rainbow.jpg'
original_image = Image.open(image_path)

# Convert the image to a NumPy array
image_data = np.array(original_image)
```

```python
height, width, _ = image_data.shape

# Flatten into DataFrame
pixel_values = image_data.reshape(-1, 3)
df = pd.DataFrame(pixel_values, columns=['R', 'G', 'B'])

# Display original image
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(original_image)
plt.title('Original Image')
plt.axis('off')

# Brightness-based noise filtering
noise_threshold = 200
noise_filter = (df['R'] > noise_threshold) | (df['G'] > noise_threshold) | (df['B'] > noise_threshold)
df_cleaned = df[~noise_filter]

# Build blank cleaned image with fallback average color
cleaned_image_data = np.zeros_like(image_data)
avg_color = [np.mean(df_cleaned['R']), np.mean(df_cleaned['G']), np.mean(df_cleaned['B'])]
cleaned_image_data[:, :] = avg_color

# Restore non-noise pixels to their positions
for idx, (r, g, b) in df_cleaned.iterrows():
    y = idx // width
    x = idx % width
    cleaned_image_data[y, x] = [r, g, b]

# Show cleaned image
cleaned_image = Image.fromarray(np.uint8(cleaned_image_data))
plt.subplot(1, 2, 2)
plt.imshow(cleaned_image)
plt.title('GPT Cleaned Image')
plt.axis('off')
plt.show()
```



[58]:
```
###TASK 3
from openai import OpenAI

client = OpenAI(
 ␣
 ↪api_key="sk-proj--fdKGVc5VVk7sY3J9dgR-y5ZRVadgnqpvFSWWVJ6POiwDa4_UnCicZUCLaQaZhf3lmQ_3MSz-K
)

completion = client.chat.completions.create(
    model="gpt-4o-mini",
    store=True,
    messages=[
      {
  "role": "user",
  "content": """MAKE EXECUTABLE PYTHON CODE FOR THE FOLLOWING: Task 3: KMeans␣
 ↪Clustering with scikit-learn Objective In this task, you'll apply clustering␣
 ↪techniques to the image dataset to identify and group pixels with similar␣
 ↪properties. The main goal is to learn how clustering can reveal patterns in␣
 ↪data and segment the image into distinct regions based on color and position.
 ↪ Steps to Follow Standardize the Data: Choose an appropriate scaler from␣
 ↪scikit-learn to standardize the pixel data, ensuring all features contribute␣
 ↪equally to the clustering process. You may use any scaler that suits your␣
 ↪needs (e.g., StandardScaler, MinMaxScaler). Hint: Experimenting with␣
 ↪different scalers can help you understand their impact on clustering results.
 ↪ Perform KMeans Clustering: Utilize KMeans from scikit-learn to cluster the␣
 ↪dataset into groups. Select the number of clusters based on your analysis or␣
 ↪experimentation. Note: Clustering helps in understanding how data points␣
 ↪(pixels) relate based on their features (x, y, R, G, B). Add Cluster Labels␣
 ↪to the DataFrame: Assign the cluster labels to each pixel and append them to␣
 ↪the DataFrame for further analysis and visualization. Visualize the␣
 ↪Clustered Data: Use any plotting library of your choice to create a␣
 ↪visualization that shows how the image is segmented into clusters. Tip:␣
 ↪Customize your plots to highlight the clusters effectively (e.g., color␣
 ↪coding based on cluster labels)."""
}

    ]
)

print(completion.choices[0].message)
print(completion.choices[0].message.content)
```

```
ChatCompletionMessage(content='Below is an executable Python code that performs
KMeans clustering on an image dataset using scikit-learn. It standardizes the
pixel data, applies the KMeans clustering algorithm, and visualizes the
```

segmented image based on the clustering. Make sure you have the required
libraries installed (`numpy`, `matplotlib`, `scikit-learn`, and `PIL` for image
handling).\n\n```python\nimport numpy as np\nimport matplotlib.pyplot as
plt\nfrom sklearn.cluster import KMeans\nfrom sklearn.preprocessing import
StandardScaler\nfrom sklearn import datasets\nfrom PIL import Image\n\ndef
load_image(image_path):\n   """Load an image from the specified path and
convert it to a NumPy array."""\n   img = Image.open(image_path)\n   img =
img.resize((100, 100)) # Resize for simplicity (can change this)\n   img_array
= np.array(img)\n   return img_array\n\ndef preprocess_image(img_array):\n
"""Reshape the image data and standardize it for clustering."""\n   h, w, c =
img_array.shape\n   # Reshape to a 2D array: (pixels, color channels)\n
pixel_data = img_array.reshape(-1, c)\n   # Standardize the pixel values\n
scaler = StandardScaler()\n   pixel_data_scaled =
scaler.fit_transform(pixel_data)\n   return pixel_data_scaled, h, w\n\ndef
apply_kmeans(pixel_data_scaled, n_clusters=5):\n   """Apply KMeans clustering
to the standardized pixel data."""\n   kmeans = KMeans(n_clusters=n_clusters,
random_state=42)\n   kmeans.fit(pixel_data_scaled)\n   return
kmeans.labels_\n\ndef visualize_clusters(img_array, labels, num_clusters):\n
"""Visualize the clustered image."""\n   # Reshape labels back to image height
and width\n   clustered_image = labels.reshape(img_array.shape[0],
img_array.shape[1])\n   \n   plt.figure(figsize=(10, 5))\n   \n   # Original
Image\n   plt.subplot(1, 2, 1)\n   plt.title(\'Original Image\')\n
plt.imshow(img_array)\n   plt.axis(\'off\')\n   \n   # Clustered Image\n
plt.subplot(1, 2, 2)\n   plt.title(f\'Clustered Image (K={num_clusters})\')\n
plt.imshow(clustered_image, cmap=\'viridis\')\n   plt.axis(\'off\')\n\n
plt.tight_layout()\n   plt.show()\n\ndef main(image_path, n_clusters):\n   #
Load and process the image\n   img_array = load_image(image_path)\n
pixel_data_scaled, h, w = preprocess_image(img_array)\n   \n   # Apply KMeans
clustering\n   labels = apply_kmeans(pixel_data_scaled, n_clusters)\n   \n
Visualize the results\n   visualize_clusters(img_array, labels,
n_clusters)\n\nif __name__ == "__main__":\n   # Path to the image file (Update
this path to your image)\n   image_path = \'path/to/your/image.jpg\'\n
n_clusters = 5 # Adjust the number of clusters as needed\n   main(image_path,
n_clusters)\n```\n\n### Instructions:\n1. **Install Required Libraries**: Make
sure you have the necessary libraries installed. You can install them using
pip:\n   ```bash\n   pip install numpy matplotlib scikit-learn Pillow\n   ```\n
\n2. **Update the Image Path**: Replace `\'path/to/your/image.jpg\'` with the
actual path to your image file.\n\n3. **Run the Code**: Execute the script to
perform KMeans clustering on the image, visualize the original and clustered
images.\n\nThe code includes the following functionalities:\n- Load an image and
resize it for simplicity.\n- Preprocess the image by reshaping and standardizing
the pixel data.\n- Apply KMeans clustering to segment the image based on
color.\n- Visualize both the original and clustered images for
comparison.\n\nFeel free to experiment with different numbers of clusters to see
how the segmentation changes!', refusal=None, role='assistant', annotations=[],
audio=None, function_call=None, tool_calls=None)
Below is an executable Python code that performs KMeans clustering on an image
dataset using scikit-learn. It standardizes the pixel data, applies the KMeans

clustering algorithm, and visualizes the segmented image based on the clustering. Make sure you have the required libraries installed (`numpy`, `matplotlib`, `scikit-learn`, and `PIL` for image handling).

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn import datasets
from PIL import Image

def load_image(image_path):
    """Load an image from the specified path and convert it to a NumPy array."""
    img = Image.open(image_path)
    img = img.resize((100, 100))  # Resize for simplicity (can change this)
    img_array = np.array(img)
    return img_array

def preprocess_image(img_array):
    """Reshape the image data and standardize it for clustering."""
    h, w, c = img_array.shape
    # Reshape to a 2D array: (pixels, color channels)
    pixel_data = img_array.reshape(-1, c)
    # Standardize the pixel values
    scaler = StandardScaler()
    pixel_data_scaled = scaler.fit_transform(pixel_data)
    return pixel_data_scaled, h, w

def apply_kmeans(pixel_data_scaled, n_clusters=5):
    """Apply KMeans clustering to the standardized pixel data."""
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    kmeans.fit(pixel_data_scaled)
    return kmeans.labels_

def visualize_clusters(img_array, labels, num_clusters):
    """Visualize the clustered image."""
    # Reshape labels back to image height and width
    clustered_image = labels.reshape(img_array.shape[0], img_array.shape[1])

    plt.figure(figsize=(10, 5))

    # Original Image
    plt.subplot(1, 2, 1)
    plt.title('Original Image')
    plt.imshow(img_array)
    plt.axis('off')
```

```
    # Clustered Image
    plt.subplot(1, 2, 2)
    plt.title(f'Clustered Image (K={num_clusters})')
    plt.imshow(clustered_image, cmap='viridis')
    plt.axis('off')

    plt.tight_layout()
    plt.show()

def main(image_path, n_clusters):
    # Load and process the image
    img_array = load_image(image_path)
    pixel_data_scaled, h, w = preprocess_image(img_array)

    # Apply KMeans clustering
    labels = apply_kmeans(pixel_data_scaled, n_clusters)

    # Visualize the results
    visualize_clusters(img_array, labels, n_clusters)

if __name__ == "__main__":
    # Path to the image file (Update this path to your image)
    image_path = 'path/to/your/image.jpg'
    n_clusters = 5  # Adjust the number of clusters as needed
    main(image_path, n_clusters)
```

Instructions:
1. **Install Required Libraries**: Make sure you have the necessary libraries installed. You can install them using pip:
   ```bash
   pip install numpy matplotlib scikit-learn Pillow
   ```

2. **Update the Image Path**: Replace `'path/to/your/image.jpg'` with the actual path to your image file.

3. **Run the Code**: Execute the script to perform KMeans clustering on the image, visualize the original and clustered images.

The code includes the following functionalities:
- Load an image and resize it for simplicity.
- Preprocess the image by reshaping and standardizing the pixel data.
- Apply KMeans clustering to segment the image based on color.
- Visualize both the original and clustered images for comparison.

Feel free to experiment with different numbers of clusters to see how the segmentation changes!

[60]:
```python
###TASK 4
from openai import OpenAI

client = OpenAI(
 ␣
 ↪api_key="sk-proj--fdKGVc5VVk7sY3J9dgR-y5ZRVadgnqpvFSWWVJ6POiwDa4_UnCicZUCLaQaZhf3lmQ_3MSz-K
)

completion = client.chat.completions.create(
    model="gpt-4o-mini",
    store=True,
    messages=[
      {
  "role": "user",
  "content": """MAKE EXECUTABLE PYTHON CODE FOR THE FOLLOWING: Task 4: Custom␣
 ↪Clustering Algorithm with PyTorch Objective In this task, you'll take a step␣
 ↪beyond pre-built libraries and implement your own clustering algorithm using␣
 ↪PyTorch. This exercise will help you understand the mechanics of clustering␣
 ↪and give you a deeper appreciation for how these algorithms work under the␣
 ↪hood. Steps to Follow Prepare the Data: Ensure that the data is in a format␣
 ↪suitable for PyTorch (i.e., convert the relevant DataFrame columns to␣
 ↪PyTorch tensors). Scale the features as needed. You can apply any scaling or␣
 ↪normalization strategy you find useful. Initialize Centroids: Randomly␣
 ↪select initial centroids from the dataset. The number of clusters should be␣
 ↪chosen based on your analysis (e.g., 8 clusters). Implement the Clustering␣
 ↪Algorithm: Create a loop for a set number of iterations: Calculate Distances:
 ↪ Compute the distance from each data point to each centroid. Assign Labels:␣
 ↪Assign each data point to the nearest centroid. Update Centroids: Recompute␣
 ↪each centroid as the mean of all points assigned to it. Hint: Use torch.
 ↪cdist() for distance calculation and torch.mean() for centroid updates. Add␣
 ↪Cluster Labels to the DataFrame: Convert the computed cluster labels from␣
 ↪PyTorch tensors back to a format that can be added to the DataFrame for␣
 ↪visualization. Visualize the Clusters: Plot the clustered image data to show␣
 ↪how the pixels are grouped. Use any visualization library you prefer."""
}

    ]
)

print(completion.choices[0].message)
print(completion.choices[0].message.content)
```

```
ChatCompletionMessage(content="Below is an implementation of a custom clustering
algorithm using PyTorch. The algorithm follows the steps you've outlined,
including data preparation, initializing centroids, and iteratively updating the
clusters. For demonstration purposes, I will use synthetic data, but you can
adjust the data loading section to use your specific dataset.\n\nMake sure you
have the required libraries installed:\n\n```bash\npip install numpy pandas
```

matplotlib torch\n```\n\nHere’s the complete executable
code:\n\n```python\nimport pandas as pd\nimport numpy as np\nimport
torch\nimport matplotlib.pyplot as plt\nfrom sklearn.datasets import
make_blobs\nfrom sklearn.preprocessing import StandardScaler\n\n# Custom
Clustering Algorithm Implementation\nclass CustomKMeans:\n   def __init__(self,
n_clusters=8, max_iters=100):\n   self.n_clusters = n_clusters\n
self.max_iters = max_iters\n   self.centroids = None\n   self.labels =
None\n\n   def fit(self, X):\n   # Device configuration\n   device =
torch.device('cuda' if torch.cuda.is_available() else 'cpu')\n\n   #
Convert the data to PyTorch tensor\n   X_tensor =
torch.FloatTensor(X).to(device)\n   \n   # Initialize centroids
randomly\n   random_indices = np.random.choice(X.shape[0], self.n_clusters,
replace=False)\n   self.centroids = X_tensor[random_indices]\n\n   for
_ in range(self.max_iters):\n   # Calculate distances\n
distances = torch.cdist(X_tensor, self.centroids.unsqueeze(0), p=2)\n
\n   # Assign labels based on nearest centroid\n   self.labels
= torch.argmin(distances, dim=1)\n   \n   # Update centroids\n
for i in range(self.n_clusters):\n   if (self.labels == i).any():
# Check if there are points assigned to this centroid\n
self.centroids[i] = X_tensor[self.labels == i].mean(dim=0)\n\n   # Convert
labels back to NumPy array for convenience\n   self.labels =
self.labels.cpu().numpy()\n\n   def predict(self, X):\n   # Similar to the
fit method but without updating centroids\n   device = torch.device('cuda'
if torch.cuda.is_available() else 'cpu')\n   X_tensor =
torch.FloatTensor(X).to(device)\n   distances = torch.cdist(X_tensor,
self.centroids.unsqueeze(0), p=2)\n   return torch.argmin(distances,
dim=1).cpu().numpy()\n\n\n# Generate synthetic data\nX, y =
make_blobs(n_samples=500, centers=8, cluster_std=0.60, random_state=0)\nscaler =
StandardScaler()\nX_scaled = scaler.fit_transform(X)\n\n# Create and fit the
clustering model\nkmeans = CustomKMeans(n_clusters=8,
max_iters=100)\nkmeans.fit(X_scaled)\n\n# Add cluster labels to DataFrame\ndf =
pd.DataFrame(X, columns=['Feature 1', 'Feature 2'])\ndf['Cluster'] =
kmeans.labels\n\n# Visualize the clustering results\nplt.figure(figsize=(10,
6))\nplt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=kmeans.labels,
cmap='viridis')\nplt.scatter(kmeans.centroids.cpu().numpy()[:, 0],
kmeans.centroids.cpu().numpy()[:, 1], s=300, c='red', marker='X',
label='Centroids')\nplt.title('Custom K-Means Clustering')\nplt.xlabel('Feature
1')\nplt.ylabel('Feature 2')\nplt.legend()\nplt.show()\n```\n\n###
Explanation:\n\n1. **Data Preparation**: We use `make_blobs` to generate
synthetic data and `StandardScaler` to scale it.\n\n2. **CustomKMeans Class**:
This class implements the K-Means algorithm:\n   - **Initialization**:
`__init__` sets the number of clusters and maximum iterations.\n   - **Fit
Method**: This method performs the K-Means clustering:\n   - Initializes
centroids randomly.\n   - Computes distances using `torch.cdist`.\n   -
Assigns each point to the nearest centroid.\n   - Updates centroids based on
assigned labels.\n   - **Predict Method**: Outputs cluster labels without
changing centroids.\n\n3. **Visualization**: Finally, we create a scatter plot
that shows the clustered data points and highlights the centroids.\n\n###

Note:\n- Make sure the computing environment (CPU or CUDA-capable GPU) is set up correctly to leverage PyTorch for tensor operations. The code automatically selects the device.\n- Adjust the number of clusters (`n_clusters`) as needed based on your data and requirements.", refusal=None, role='assistant', annotations=[], audio=None, function_call=None, tool_calls=None)
Below is an implementation of a custom clustering algorithm using PyTorch. The algorithm follows the steps you've outlined, including data preparation, initializing centroids, and iteratively updating the clusters. For demonstration purposes, I will use synthetic data, but you can adjust the data loading section to use your specific dataset.

Make sure you have the required libraries installed:

```bash
pip install numpy pandas matplotlib torch
```

Here's the complete executable code:

```python
import pandas as pd
import numpy as np
import torch
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

# Custom Clustering Algorithm Implementation
class CustomKMeans:
    def __init__(self, n_clusters=8, max_iters=100):
        self.n_clusters = n_clusters
        self.max_iters = max_iters
        self.centroids = None
        self.labels = None

    def fit(self, X):
        # Device configuration
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

        # Convert the data to PyTorch tensor
        X_tensor = torch.FloatTensor(X).to(device)

        # Initialize centroids randomly
        random_indices = np.random.choice(X.shape[0], self.n_clusters,
replace=False)
        self.centroids = X_tensor[random_indices]

        for _ in range(self.max_iters):
```

```
            # Calculate distances
            distances = torch.cdist(X_tensor, self.centroids.unsqueeze(0), p=2)

            # Assign labels based on nearest centroid
            self.labels = torch.argmin(distances, dim=1)

            # Update centroids
            for i in range(self.n_clusters):
                if (self.labels == i).any():  # Check if there are points
assigned to this centroid
                    self.centroids[i] = X_tensor[self.labels == i].mean(dim=0)

        # Convert labels back to NumPy array for convenience
        self.labels = self.labels.cpu().numpy()

    def predict(self, X):
        # Similar to the fit method but without updating centroids
        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        X_tensor = torch.FloatTensor(X).to(device)
        distances = torch.cdist(X_tensor, self.centroids.unsqueeze(0), p=2)
        return torch.argmin(distances, dim=1).cpu().numpy()


# Generate synthetic data
X, y = make_blobs(n_samples=500, centers=8, cluster_std=0.60, random_state=0)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Create and fit the clustering model
kmeans = CustomKMeans(n_clusters=8, max_iters=100)
kmeans.fit(X_scaled)

# Add cluster labels to DataFrame
df = pd.DataFrame(X, columns=['Feature 1', 'Feature 2'])
df['Cluster'] = kmeans.labels

# Visualize the clustering results
plt.figure(figsize=(10, 6))
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=kmeans.labels, cmap='viridis')
plt.scatter(kmeans.centroids.cpu().numpy()[:, 0],
kmeans.centroids.cpu().numpy()[:, 1], s=300, c='red', marker='X',
label='Centroids')
plt.title('Custom K-Means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

```
### Explanation:

1. **Data Preparation**: We use `make_blobs` to generate synthetic data and
`StandardScaler` to scale it.

2. **CustomKMeans Class**: This class implements the K-Means algorithm:
   - **Initialization**: `__init__` sets the number of clusters and maximum
iterations.
   - **Fit Method**: This method performs the K-Means clustering:
     - Initializes centroids randomly.
     - Computes distances using `torch.cdist`.
     - Assigns each point to the nearest centroid.
     - Updates centroids based on assigned labels.
   - **Predict Method**: Outputs cluster labels without changing centroids.

3. **Visualization**: Finally, we create a scatter plot that shows the clustered
data points and highlights the centroids.

### Note:
- Make sure the computing environment (CPU or CUDA-capable GPU) is set up
correctly to leverage PyTorch for tensor operations. The code automatically
selects the device.
- Adjust the number of clusters (`n_clusters`) as needed based on your data and
requirements.
```

[ ]: