Image Compression & Classificaiton Prediction

Team 1 Final Project Report

Arthur: Lydia Niu, Ryan Kim, Stuart Arief, Colin Pham, Jonathan Chun

Introduction

After compressing images using k-means clustering, we will turn compression into a prediction problem. We are using k-means clustering to compress images, measure how much quality is lost, and then use a supervised model (KNN) to predict which images belong to a certain category.

Question:

Since images vary in how compressible they are, can we measure and predict compression quality? Can we predict which images are easier to compress?

Dataset

Original images:
[https://drive.google.com/drive/folders/1ryd2Wb7_p6aZk9N13_B5Mxxn2PMu31sl?usp=drive_link]

Compressed files:
[https://drive.google.com/drive/folders/1PNfPuXhBIHCHwAfNh_zxUZie5xAFwV-2?usp=drive_link]

Presentation Slides link

Slides:
[https://docs.google.com/presentation/d/11lTfTXgLVy_cAzzh6rujusyNi2xRTISkRa52MyvfTe8/edit?usp=drive_link]

Video: [https://drive.google.com/file/d/1tEYF1g7hW6OKygnK4xraPEfYK8RChDGU/view?usp=drive_link]

Technique and Explanation

K-means clustering: We are using K-means clustering in our project in order to compress images. This is achieved by clustering individual pixels within a photo into clusters based on how similar their RGB values are. Once these clusters are computed, each pixel is replaced by their respective cluster's centroid color. Effectively, this reduces the total amount of colors within the photo and reduces compresses the quality of the photo.

K-Nearest Neighbors (KNN): We use KNN to improve compression performance for architecture and landscape images. These image categories showed higher error at low K values due to high edge density and high amounts of color. Using this method allowed us to improve image quality in terms of both MSE and SSIM.

Feature Selection: Forward Selection: In our project, we started with a program to initially test compression and whether or not we chose the right parameters. Additionally, we began to look at how the program ran differently amongst the different image categories. Backword Elimination: We first started out by compressing all images at a certain value of k. Afterwards, we could look and analyze to see how that value of K impacted the photo and adjusted k values per category to bet suit each category.

MSE and SSIM: Mean Square Error: We use mean square error in order to analyze the quality of compression. If there is a high MSE when comparing the compressed image, we know that the image has lost a lot of colors/quality. SSIM: SSIM is used to measure the difference in brightness, contrast, and image structure (textures and shapes). This is used in the project to measure the perceived quality of our photos.

Compression - K-means clustering

```
import cv2
import numpy as np
import matplotlib.pyplot as plt



# ello - READ THIS

# make sure cv2 is downloaded. Change file input/output names as well
as K values below
# sometimes it might take a fat minute to load
#----------------------------------------------------------------------
---------------------------
image_path = "originals/architecture/arc1.jpg"  # CHANGE THIS TO THE
CORRECT FILE NAME DO NOT FORGET
output_path = "compressed_image.jpg"
K = 32  # MODIFY K
image = cv2.imread(image_path)
if image is None:
    raise FileNotFoundError(f"Could not load image at {image_path}")
#----------------------------------------------------------------------
---------------------------

# image dimensions
height, width = image.shape[:2]
```

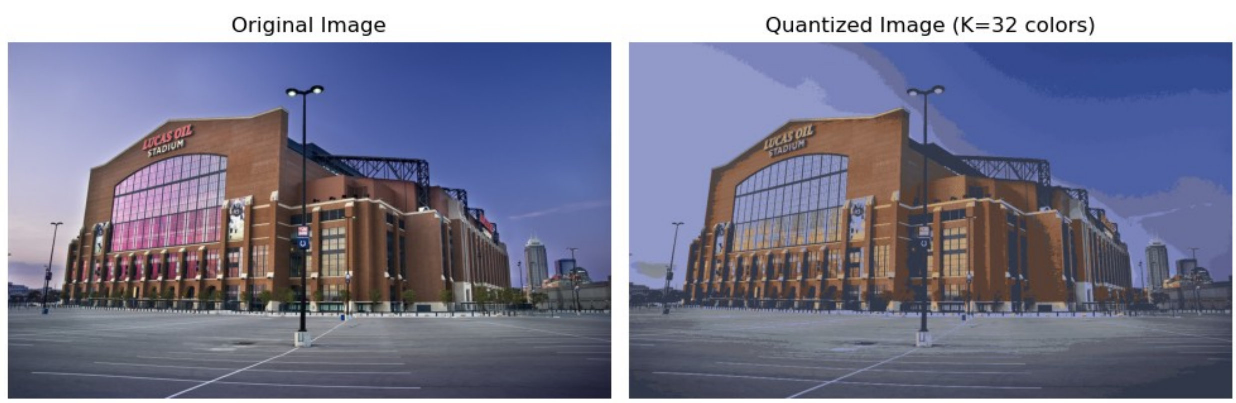
```
# turn into 2d array of RBG values
flat_pixels = image.reshape(-1, 3).astype(np.float32)
num_pixels = flat_pixels.shape[0]

# choose k
random_indices = np.random.choice(num_pixels, size=K, replace=False)
representative_colors = flat_pixels[random_indices]
diff = flat_pixels[:, None, :] - representative_colors[None, :, :]
distances_sq = np.sum(diff**2, axis=2)

# Distance to nearest color calculations
nearest_color_idx = np.argmin(distances_sq, axis=1)
quantized_flat_pixels = representative_colors[nearest_color_idx]
quantized_flat_pixels =
np.rint(quantized_flat_pixels).astype(np.uint8)
quantized_image = quantized_flat_pixels.reshape(height, width, 3)

cv2.imwrite(output_path, quantized_image)
print(f"Compressed image saved as {output_path} with {K} colors.")
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
quantized_image_rgb = cv2.cvtColor(quantized_image, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_rgb)
plt.title("Original Image")
plt.axis('off')
plt.subplot(1, 2, 2)
plt.imshow(quantized_image_rgb)
plt.title(f"Quantized Image (K={K} colors)")
plt.axis('off')
plt.tight_layout()
plt.show()
```

```
Compressed image saved as compressed_image.jpg with 32 colors.
```



EDA - Error Metrics Summary (MSE + SSIM)

I created reproducible functions to calculate MSE, SSIM, and generate the table of the metrics summary for each category

```
import cv2
import numpy as np
import os
import glob
import pandas as pd
from skimage.metrics import structural_similarity as ssim

# --- Metric Calculation Function ---
def calculate_metrics(original_img, compressed_img):
    """
    Calculates MSE and SSIM between the original and compressed
images.
    """

    # Ensure images have the same shape
    if original_img.shape != compressed_img.shape:
        raise ValueError("Original and compressed images must have the
same dimensions.")

    # 1. Mean Squared Error (MSE)
    error = np.sum((original_img.astype("float") -
compressed_img.astype("float")) ** 2)
    N = float(original_img.shape[0] * original_img.shape[1] *
original_img.shape[2])
    mse = error / N

    # 2. Structural Similarity Index Measure (SSIM)
    original_gray = cv2.cvtColor(original_img, cv2.COLOR_BGR2GRAY)
    compressed_gray = cv2.cvtColor(compressed_img, cv2.COLOR_BGR2GRAY)

    # FIX: Only capture the SSIM score, as your skimage version is not
returning the tuple (score, diff_map).
    ssim_score = ssim(original_gray, compressed_gray, data_range=255,
channel_axis=None)

    return mse, ssim_score

#
----------------------------------------------------------------------
-----------

def generate_category_matrix(category_name, prefix, k_values,
originals_dir="originals", compressed_dir="compressed"):
    """
```

```
    Generates the error matrix for a single image category, sorted by
image ID and K-Value.

    Returns:
        pandas.DataFrame: The resulting error matrix for the category.
    """

    original_path = os.path.join(originals_dir, category_name)
    compressed_mod_path = os.path.join(compressed_dir,
f"{category_name}_mod")

    # Finds files (e.g., 'arc1.jpg', 'arc2.jpg')
    original_files = glob.glob(os.path.join(original_path,
f"{prefix}*.jpg"))

    if not original_files:
        print(f"Warning: No original files found for category
'{category_name}'.")
        return pd.DataFrame()

    results = []

    print(f"\n--- Processing Category: {category_name.upper()} ---")

    for og_file_path in original_files:
        base_filename = os.path.basename(og_file_path)
        image_id = os.path.splitext(base_filename)[0]

        og_image = cv2.imread(og_file_path)
        if og_image is None:
            print(f"Skipping {base_filename}: could not load original
image.")
            continue

        for K in k_values:
            # Assumes compressed filename: arc1_K4.jpg, arc1_K8.jpg,
etc.
            compressed_filename = f"{image_id}_K{K}.jpg"
            compressed_file_path = os.path.join(compressed_mod_path,
compressed_filename)

            comp_image = cv2.imread(compressed_file_path)

            if comp_image is None:
                print(f"  - Missing compressed file for {image_id} at
K={K}. Skipping.")
                continue

            mse, ssim_score = calculate_metrics(og_image, comp_image)
```

```
            # Store the numeric part of the ID (e.g., 1, 2, 3) for
sorting
            image_num_id = int(''.join(filter(str.isdigit, image_id)))

            results.append({
                'Image_ID': image_id,
                'Image_Num_ID': image_num_id, # Used for sorting
(Requirement 1)
                'K_Value': K,
                'MSE': round(mse, 4),
                'SSIM': round(ssim_score, 4)
            })

    df = pd.DataFrame(results)

    # Requirement 1 & 2: Sort by numeric Image ID (ascending) then by
K_Value (ascending)
    df = df.sort_values(by=['Image_Num_ID',
'K_Value']).drop(columns=['Image_Num_ID']).reset_index(drop=True)

    return df
#
----------------------------------------------------------------------
-----------

# All K values will be checked
K_VALUES_TO_CHECK = [4, 8, 16]

CATEGORIES = [
    {'name': 'architecture', 'prefix': 'arc'},
    {'name': 'landscapes', 'prefix': 'land'},
    {'name': 'portraits', 'prefix': 'port'},
    {'name': 'still_life', 'prefix': 'life'},
    {'name': 'urban', 'prefix': 'urb'},
]

# --- Run the generation and collect matrices ---
all_matrices = {}
overall_averages = {}

for cat in CATEGORIES:
    df_matrix = generate_category_matrix(
        category_name=cat['name'],
        prefix=cat['prefix'],
        k_values=K_VALUES_TO_CHECK
    )

    if not df_matrix.empty:
        # Calculate the overall average for the category (Requirement
```

```python
3)
        avg_mse = df_matrix['MSE'].mean()
        avg_ssim = df_matrix['SSIM'].mean()
        overall_averages[cat['name']] = {'Avg_MSE': round(avg_mse, 4),
'Avg_SSIM': round(avg_ssim, 4)}

    all_matrices[cat['name']] = df_matrix

# --- Display the results and summaries ---

print("\n\n#####################################################")
print("#### DETAILED ERROR MATRIX SUMMARY BY CATEGORY ####")
print("#####################################################")

# Display the detailed matrices
for category, matrix in all_matrices.items():
    print(f"\n\n--- DETAILED MATRIX: {category.upper()}
(K={K_VALUES_TO_CHECK}) ---")
    if not matrix.empty:
        print(matrix.to_markdown(index=False))
    else:
        print("No data available for this category.")

# Display the overall summary (Requirement 3)
print("\n" + "="*53)
print("### CATEGORY OVERALL PERFORMANCE SUMMARY ###")
print("="*53)

# Create a DataFrame for the summary table
summary_df = pd.DataFrame.from_dict(overall_averages, orient='index')
summary_df.index.name = "Category"

# Rename columns for clarity
summary_df.columns = ['Average MSE', 'Average SSIM']

# Display the summary table
print(summary_df.to_markdown())
print("="*53)
```

```
--- Processing Category: ARCHITECTURE ---

--- Processing Category: LANDSCAPES ---

--- Processing Category: PORTRAITS ---

--- Processing Category: STILL_LIFE ---

--- Processing Category: URBAN ---
```

```
#####################################################
#### DETAILED ERROR MATRIX SUMMARY BY CATEGORY ####
#####################################################


--- DETAILED MATRIX: ARCHITECTURE (K=[4, 8, 16]) ---
| Image_ID    |   K_Value |        MSE |   SSIM |
|:------------|----------:|----------:|-------:|
| arc1        |         4 |  446.841  | 0.8022 |
| arc1        |         8 |  191.564  | 0.8498 |
| arc1        |        16 |  195.554  | 0.8889 |
| arc2        |         4 |  555.808  | 0.8346 |
| arc2        |         8 |  143.579  | 0.8702 |
| arc2        |        16 |  346.135  | 0.8348 |
| arc3        |         4 | 1018.65   | 0.7189 |
| arc3        |         8 |  427.428  | 0.7627 |
| arc3        |        16 |  431.874  | 0.7416 |
| arc4        |         4 |  413.322  | 0.8439 |
| arc4        |         8 |  132.275  | 0.8809 |
| arc4        |        16 |  353.574  | 0.8319 |
| arc5        |         4 |  362.734  | 0.8497 |
| arc5        |         8 |  174.036  | 0.9047 |
| arc5        |        16 | 2464.33   | 0.7975 |
| arc6        |         4 |  202.646  | 0.9477 |
| arc6        |         8 |   82.0329 | 0.9473 |
| arc6        |        16 |  110.287  | 0.9538 |
| arc7        |         4 |  343.166  | 0.9065 |
| arc7        |         8 |  171.633  | 0.917  |
| arc7        |        16 |  119.472  | 0.927  |
| arc8        |         4 |  249.603  | 0.915  |
| arc8        |         8 |   83.6793 | 0.9101 |
| arc8        |        16 |  139.009  | 0.9168 |
| arc9        |         4 |  394.538  | 0.8901 |
| arc9        |         8 |  127.892  | 0.8807 |
| arc9        |        16 |  574.217  | 0.9103 |
| arc10       |         4 |  434.153  | 0.8406 |
| arc10       |         8 |  164.698  | 0.8709 |
| arc10       |        16 |  356.077  | 0.8456 |
| arc11       |         4 |  459.265  | 0.7478 |
| arc11       |         8 |  190.657  | 0.7776 |
| arc11       |        16 |  404.171  | 0.808  |
| arc12       |         4 |  199.721  | 0.9334 |
| arc12       |         8 |   83.405  | 0.9323 |
| arc12       |        16 |  181.184  | 0.9371 |
| arc13       |         4 |  355.013  | 0.7088 |
| arc13       |         8 |  127.497  | 0.7802 |
| arc13       |        16 |  207.693  | 0.8186 |
| arc14       |         4 |  276.184  | 0.8923 |
| arc14       |         8 |   98.2728 | 0.8994 |
```

```
| arc14       |         16 |  657.156  | 0.8554 |
| arc15       |          4 |  972.692  | 0.8601 |
| arc15       |          8 |  164.417  | 0.885  |
| arc15       |         16 |   85.5812 | 0.9112 |
| arc16       |          4 |  509.863  | 0.5399 |
| arc16       |          8 |  187.162  | 0.7577 |
| arc16       |         16 |  242.166  | 0.8098 |
| arc17       |          4 |  576.223  | 0.8036 |
| arc17       |          8 |  248.878  | 0.8277 |
| arc17       |         16 |  640.105  | 0.8147 |
| arc18       |          4 |  188.461  | 0.8383 |
| arc18       |          8 |   66.104  | 0.875  |
| arc18       |         16 |   78.4862 | 0.8613 |
| arc19       |          4 |  514.023  | 0.759  |
| arc19       |          8 |  166.888  | 0.8445 |
| arc19       |         16 | 1373.88   | 0.8184 |
| arc20       |          4 |  435.155  | 0.7706 |
| arc20       |          8 |  159.951  | 0.8292 |
| arc20       |         16 |  176.175  | 0.812  |
| arc21       |          4 |  285.76   | 0.8645 |
| arc21       |          8 |   66.5744 | 0.9146 |
| arc21       |         16 |  140.904  | 0.9042 |
| arc22       |          4 |  835.539  | 0.7049 |
| arc22       |          8 |  224.847  | 0.7108 |
| arc22       |         16 |  213.342  | 0.8105 |
| arc23       |          4 |  681.561  | 0.7564 |
| arc23       |          8 |  275.826  | 0.8132 |
| arc23       |         16 |  606.658  | 0.7712 |


--- DETAILED MATRIX: LANDSCAPES (K=[4, 8, 16]) ---
| Image_ID    |   K_Value |     MSE |   SSIM |
|:------------|----------:|--------:|-------:|
| land1       |          4 | 563.374 | 0.7388 |
| land1       |          8 | 208.672 | 0.7925 |
| land1       |         16 | 209.146 | 0.8374 |
| land2       |          4 | 408.682 | 0.793  |
| land2       |          8 | 199.073 | 0.8443 |
| land2       |         16 | 204.341 | 0.8741 |
| land3       |          4 | 479.697 | 0.7219 |
| land3       |          8 | 205.674 | 0.7386 |
| land3       |         16 | 330.625 | 0.7869 |
| land4       |          4 | 326.348 | 0.7377 |
| land4       |          8 | 146.574 | 0.8264 |
| land4       |         16 | 172.26  | 0.8332 |
| land5       |          4 | 601.419 | 0.6661 |
| land5       |          8 | 237.406 | 0.6991 |
| land5       |         16 | 729.124 | 0.724  |
| land6       |          4 | 390.656 | 0.8207 |
```

land6	8	170.239	0.8474
land6	16	203.1	0.8841
land7	4	398.558	0.7098
land7	8	176.113	0.7755
land7	16	360.755	0.7684
land8	4	270.096	0.8435
land8	8	126.477	0.86
land8	16	233.395	0.8732
land9	4	659.4	0.7312
land9	8	198.16	0.7947
land9	16	412.424	0.7669
land10	4	596.23	0.8181
land10	8	239.588	0.8585
land10	16	325.061	0.8531
land11	4	624.978	0.7324
land11	8	264.328	0.8175
land11	16	331.953	0.8393
land12	4	235.938	0.8101
land12	8	108.436	0.8862
land12	16	161.527	0.8972
land13	4	313.572	0.7892
land13	8	132.51	0.8456
land13	16	146.846	0.8497
land14	4	610.758	0.6185
land14	8	254.812	0.7558
land14	16	376.555	0.7752
land15	4	604.366	0.6775
land15	8	268.674	0.7614
land15	16	577.159	0.8121
land16	4	592.262	0.7204
land16	8	191.845	0.7981
land16	16	384.486	0.797
land17	4	756.215	0.6702
land17	8	336.205	0.8088
land17	16	496.636	0.7939
land18	4	342.925	0.6572
land18	8	160.235	0.7816
land18	16	198.111	0.7519
land19	4	626.126	0.6752
land19	8	273.054	0.7933
land19	16	262.837	0.7796
land20	4	913.97	0.6596
land20	8	466.718	0.7506
land20	16	596.195	0.7348
land21	4	604.907	0.61
land21	8	241.031	0.6911
land21	16	274.543	0.7393
land22	4	312.867	0.7611
land22	8	130.045	0.8111

land22	16	220.877	0.8736
land23	4	265.699	0.7655
land23	8	118.121	0.8523
land23	16	295.373	0.85

--- DETAILED MATRIX: PORTRAITS (K=[4, 8, 16]) ---

Image_ID	K_Value	MSE	SSIM
port1	4	162.03	0.5846
port1	8	76.1883	0.7968
port1	16	250.05	0.8227
port2	4	463.045	0.8098
port2	8	186.606	0.8449
port2	16	354.596	0.8617
port3	4	272.476	0.5892
port3	8	109.725	0.783
port3	16	140.083	0.7976
port4	4	399.537	0.722
port4	8	148.875	0.8008
port4	16	322.051	0.8444
port5	4	699.212	0.7379
port5	8	352.28	0.748
port5	16	376.903	0.7745
port6	4	177.19	0.8963
port6	8	80.4583	0.9029
port6	16	104.349	0.91
port7	4	480.368	0.7981
port7	8	249.261	0.8246
port7	16	395.718	0.839
port8	4	114.147	0.7789
port8	8	45.3917	0.8429
port8	16	529.952	0.893
port9	4	553.168	0.8061
port9	8	268.342	0.8147
port9	16	316.976	0.8381
port10	4	404.741	0.8282
port10	8	127.783	0.8293
port10	16	247.704	0.8128
port11	4	301.048	0.8279
port11	8	129.274	0.8494
port11	16	206.823	0.8757
port12	4	236.581	0.6552
port12	8	115.061	0.7672
port12	16	193.313	0.7688
port13	4	410.717	0.5078
port13	8	165.002	0.7032
port13	16	303.724	0.6907
port14	4	425.006	0.6904

port14	8	170.237	0.7515
port14	16	279.537	0.7906
port15	4	551.931	0.7355
port15	8	96.644	0.7968
port15	16	108.841	0.8298
port16	4	348.358	0.7629
port16	8	147.57	0.8046
port16	16	133.327	0.8288
port17	4	537.988	0.4057
port17	8	212.649	0.7689
port17	16	636.029	0.6824
port18	4	222.776	0.8716
port18	8	94.6057	0.8929
port18	16	121.346	0.9106
port19	4	409.575	0.8418
port19	8	203.488	0.8666
port19	16	265.726	0.8481
port20	4	692.641	0.7696
port20	8	338.066	0.8122
port20	16	464.246	0.8029
port21	4	500.266	0.5975
port21	8	195.542	0.7488
port21	16	164.459	0.8573
port22	4	264.291	0.7954
port22	8	86.2904	0.8052
port22	16	108.588	0.8115

--- DETAILED MATRIX: STILL_LIFE (K=[4, 8, 16]) ---

Image_ID	K_Value	MSE	SSIM
life1	4	192.396	0.9378
life1	8	155.702	0.9269
life1	16	2137.39	0.9329
life2	4	443.094	0.4741
life2	8	184.696	0.5932
life2	16	356.31	0.8222
life3	4	420.289	0.6762
life3	8	181.974	0.7339
life3	16	208.224	0.8022
life4	4	246.379	0.777
life4	8	107.22	0.7943
life4	16	158.841	0.8564
life5	4	562.128	0.8454
life5	8	178.668	0.8705
life5	16	829.808	0.8594
life6	4	172.639	0.7139
life6	8	147.751	0.7265
life6	16	116.581	0.9384

life7	4	495.001	0.7035
life7	8	191.675	0.7398
life7	16	347.225	0.8035
life8	4	416.485	0.8887
life8	8	135.397	0.902
life8	16	160.046	0.9146
life9	4	218.733	0.5153
life9	8	69.5917	0.7782
life9	16	356.313	0.8725
life10	4	185.325	0.883
life10	8	60.0123	0.9156
life10	16	74.7789	0.936
life11	4	129.214	0.8821
life11	8	67.3523	0.9147
life11	16	121.07	0.9312
life12	4	473.779	0.6192
life12	8	180.099	0.7434
life12	16	338.358	0.8567
life13	4	253.476	0.8816
life13	8	91.962	0.8878
life13	16	351.161	0.8676
life14	4	289.172	0.8857
life14	8	102.995	0.8829
life14	16	405.145	0.904
life15	4	328.936	0.9204
life15	8	173.748	0.8584
life15	16	139.294	0.8969
life16	4	803.654	0.8566
life16	8	389.565	0.852
life16	16	563.021	0.8522
life17	4	578.604	0.6987
life17	8	231.17	0.7649
life17	16	347.274	0.7644
life18	4	165.917	0.9433
life18	8	106.191	0.9444
life18	16	268.995	0.9271
life19	4	184.204	0.8964
life19	8	88.2407	0.9226
life19	16	108.011	0.9323
life20	4	305.191	0.8233
life20	8	138.515	0.8428
life20	16	770.414	0.8437
life21	4	588.43	0.8151
life21	8	201.329	0.829
life21	16	689.129	0.8449
life22	4	534.295	0.9798
life22	8	123.583	0.9675
life22	16	175.954	0.9824
life23	4	388.718	0.8855

life23	8	180.671	0.8498
life23	16	402.23	0.9023
life24	4	399.418	0.947
life24	8	289.253	0.9548
life24	16	281.328	0.9453
life25	4	292.757	0.8482
life25	8	226.988	0.8766
life25	16	110.058	0.9187

--- DETAILED MATRIX: URBAN (K=[4, 8, 16]) ---

Image_ID	K_Value	MSE	SSIM
urb1	4	641.443	0.7035
urb1	8	257.919	0.7663
urb1	16	355.878	0.8144
urb2	4	272.218	0.6767
urb2	8	107.329	0.8322
urb2	16	84.2232	0.8933
urb3	4	512.126	0.6434
urb3	8	256.604	0.7524
urb3	16	442.223	0.8017
urb4	4	523.561	0.8107
urb4	8	216.867	0.8381
urb4	16	220.763	0.8711
urb5	4	500.777	0.8097
urb5	8	254.865	0.8586
urb5	16	251.847	0.8716
urb6	4	587.995	0.607
urb6	8	242.005	0.7342
urb6	16	578.902	0.7926
urb7	4	728.674	0.7338
urb7	8	300.158	0.8106
urb7	16	1202.98	0.7977
urb8	4	539.852	0.7562
urb8	8	239.207	0.7954
urb8	16	254.006	0.8192
urb9	4	408.803	0.6318
urb9	8	194.119	0.7351
urb9	16	916.766	0.7876
urb10	4	387.821	0.6127
urb10	8	150.403	0.7233
urb10	16	240.118	0.7944
urb11	4	367.298	0.7271
urb11	8	141.69	0.8165
urb11	16	142.497	0.8591
urb12	4	413.471	0.7642
urb12	8	153.79	0.8418
urb12	16	447.179	0.8419

```
| urb13        |         4 |  469.353  | 0.7803 |
| urb13        |         8 |  205.75   | 0.8311 |
| urb13        |        16 |  298.921  | 0.8312 |
| urb14        |         4 |  553.798  | 0.6891 |
| urb14        |         8 |  248.751  | 0.7396 |
| urb14        |        16 |  435.447  | 0.8249 |
| urb15        |         4 |  304.324  | 0.7862 |
| urb15        |         8 |  137.339  | 0.8515 |
| urb15        |        16 |  100.978  | 0.8909 |
| urb16        |         4 |  528.432  | 0.806  |
| urb16        |         8 |  225.934  | 0.8517 |
| urb16        |        16 |  618.905  | 0.8646 |
| urb17        |         4 |  342.404  | 0.6102 |
| urb17        |         8 |  177.751  | 0.7841 |
| urb17        |        16 |  164      | 0.8474 |
| urb18        |         4 |  300.209  | 0.7064 |
| urb18        |         8 |   81.2112 | 0.8143 |
| urb18        |        16 |  126.884  | 0.8513 |
| urb19        |         4 | 1063.81   | 0.7234 |
| urb19        |         8 |  491.143  | 0.7757 |
| urb19        |        16 |  703.31   | 0.809  |
| urb20        |         4 |  314.481  | 0.7711 |
| urb20        |         8 |  134.533  | 0.8133 |
| urb20        |        16 |  232.048  | 0.8701 |
| urb21        |         4 |  530.586  | 0.7686 |
| urb21        |         8 |  243.707  | 0.8359 |
| urb21        |        16 |  213.324  | 0.8359 |
| urb22        |         4 |  515.667  | 0.7569 |
| urb22        |         8 |  252.726  | 0.8169 |
| urb22        |        16 |  286.823  | 0.8464 |
| urb23        |         4 |  396.616  | 0.7466 |
| urb23        |         8 |  178.52   | 0.8728 |
| urb23        |        16 |  176.683  | 0.9011 |

=====================================================
### CATEGORY OVERALL PERFORMANCE SUMMARY ###
=====================================================
| Category     |   Average MSE |   Average SSIM |
|:-------------|--------------:|---------------:|
| architecture |       356.062 |         0.8399 |
| landscapes   |       345.744 |         0.7799 |
| portraits    |       276.527 |         0.7857 |
| still_life   |       305.194 |         0.8464 |
| urban        |       356.387 |         0.7888 |
=====================================================
```

Explanation: Error Metrics Summary (MSE & SSIM) To evaluate how well our compressed images perform at different k-values, we summarize two key metrics for every category: MSE and SSIM. MSE, or mean squared error, measures the average squared difference between the original and compressed images, so lower values mean the compressed image stays closer to the original. SSIM, or structural similarity index, compares the structure, brightness, and contrast of the two images. It ranges from 0 to 1, with values closer to 1 indicating higher similarity. Using both metrics gives a balanced understanding of pixel-level accuracy and perceptual quality.

From the category tables, we can see how each individual image behaves across different k-values. Some images show only small changes in MSE and SSIM when k increases, meaning they are easier to compress and maintain quality even at lower k. Others show large drops in SSIM or spikes in MSE at low k, meaning they require more clusters to preserve their structure or color variations. By reading across each row, we can identify consistent patterns, such as which images lose detail quickly, which ones stay stable across k, and how sensitivity differs based on image content.

At the end, the overall summary table averages MSE and SSIM at each k for the entire category. This tells us the general performance of each category instead of focusing on individual images. From this, we can see which categories compress well as a whole, which k-values give the best balance between compression and quality, and how categories differ in structural or pixel-level loss. These averages help guide a broader conclusion about the optimal k-value and which types of images are more or less compressible overall.

Optimization: EDA of K-means Clustering on Architecture

```
import cv2
import numpy as np
import os
import glob
import pandas as pd
from skimage.metrics import structural_similarity as ssim

# ---------- Metrics ----------
def calculate_metrics(original_img, compressed_img):
    # MSE
    error = np.sum((original_img.astype("float") -
compressed_img.astype("float")) ** 2)
    N = float(original_img.shape[0] * original_img.shape[1] *
original_img.shape[2])
    mse = error / N

    # SSIM (convert to grayscale)
    original_gray = cv2.cvtColor(original_img, cv2.COLOR_BGR2GRAY)
    compressed_gray = cv2.cvtColor(compressed_img, cv2.COLOR_BGR2GRAY)
    ssim_score = ssim(original_gray, compressed_gray, data_range=255,
channel_axis=None)

    return mse, ssim_score
```

```python
def generate_opt_matrix(category_name, prefix, originals_dir,
compressed_dir):
    """
    Compare originals vs *opt images for one category.
    Example:
        originals/architecture/arc1.jpg
        compressed/architecture_opt/arc1opt.jpg
    """
    original_path = os.path.join(originals_dir, category_name)
    compressed_opt_path = os.path.join(compressed_dir,
f"{category_name}_opt")

    original_files = glob.glob(os.path.join(original_path,
f"{prefix}*.jpg"))
    if not original_files:
        print(f"Warning: no originals for {category_name}")
        return pd.DataFrame()

    rows = []

    for og_file_path in original_files:
        base = os.path.basename(og_file_path)       # arc1.jpg
        image_id = os.path.splitext(base)[0]        # arc1

        og = cv2.imread(og_file_path)
        if og is None:
            print(f"Skipping {base}: cannot load original.")
            continue

        comp_name = f"{image_id}opt.jpg"            # arc1opt.jpg
        comp_path = os.path.join(compressed_opt_path, comp_name)
        comp = cv2.imread(comp_path)

        if comp is None:
            print(f"Missing optimized file: {comp_name}")
            continue

        mse, ssim_score = calculate_metrics(og, comp)

        num_part = ''.join(filter(str.isdigit, image_id))
        num_id = int(num_part) if num_part else 0

        rows.append({
            "Image_ID": image_id + "opt",
            "Image_Num_ID": num_id,
            "MSE": round(mse, 4),
            "SSIM": round(ssim_score, 4)
        })

    df = pd.DataFrame(rows)
```

```
    if df.empty:
        return df

    df =
df.sort_values("Image_Num_ID").drop(columns=["Image_Num_ID"]).reset_in
dex(drop=True)
    return df


# ----- RUN: K-means on ARCHITECTURE -----
originals_dir = r"originals"
compressed_dir = r"compressed"

arch_df = generate_opt_matrix(
    category_name="architecture",
    prefix="arc",
    originals_dir=originals_dir,
    compressed_dir=compressed_dir
)

print("--- OPTIMIZED MATRIX: ARCHITECTURE (K-means) ---")
print(arch_df.to_markdown(index=False))

print("\n--- ARCHITECTURE (K-means) AVERAGES ---")
print(f"Average MSE  : {arch_df['MSE'].mean():.4f}")
print(f"Average SSIM : {arch_df['SSIM'].mean():.4f}")
```

```
--- OPTIMIZED MATRIX: ARCHITECTURE (K-means) ---
| Image_ID    |       MSE |   SSIM |
|:------------|----------:|-------:|
| arc1opt     |   49.2148 | 0.9283 |
| arc2opt     |   30.8436 | 0.944  |
| arc3opt     |  144.153  | 0.8475 |
| arc4opt     |   36.8806 | 0.9514 |
| arc5opt     |   43.5786 | 0.949  |
| arc6opt     | 7216.81   | 0.8634 |
| arc7opt     |   34.1229 | 0.9479 |
| arc8opt     |   19.4338 | 0.9474 |
| arc9opt     |   26.5575 | 0.9424 |
| arc10opt    |   25.8229 | 0.9427 |
| arc11opt    |   55.6654 | 0.8691 |
| arc12opt    |   24.4445 | 0.9538 |
| arc13opt    |   29.7676 | 0.9126 |
| arc14opt    |   17.2027 | 0.968  |
| arc15opt    |   10.1125 | 0.9731 |
| arc16opt    |   50.811  | 0.9475 |
| arc17opt    |   60.6909 | 0.9139 |
| arc18opt    |   16.5576 | 0.9535 |
| arc19opt    |   56.7487 | 0.9217 |
| arc20opt    |   32.2402 | 0.9207 |
```

```
| arc21opt    |     8.4035 | 0.98   |
| arc22opt    |    46.2274 | 0.8992 |
| arc23opt    |    70.5238 | 0.9034 |

--- ARCHITECTURE (K-means) AVERAGES ---
Average MSE  : 352.4702
Average SSIM : 0.9296
```

Technique 2: KNN (Supervised Learning)

EDA of KNN for Landscapes

```
import cv2
import numpy as np
import os
import glob
import pandas as pd
from skimage.metrics import structural_similarity as ssim

# ---------- Metrics ----------
def calculate_metrics(original_img, compressed_img):
    # MSE
    error = np.sum((original_img.astype("float") -
compressed_img.astype("float")) ** 2)
    N = float(original_img.shape[0] * original_img.shape[1] *
original_img.shape[2])
    mse = error / N

    # SSIM (convert to grayscale)
    original_gray = cv2.cvtColor(original_img, cv2.COLOR_BGR2GRAY)
    compressed_gray = cv2.cvtColor(compressed_img, cv2.COLOR_BGR2GRAY)
    ssim_score = ssim(original_gray, compressed_gray, data_range=255,
channel_axis=None)

    return mse, ssim_score


def generate_opt_matrix(category_name, prefix, originals_dir,
compressed_dir):
    """
    Compare originals vs *opt images for one category.
    Example:
        originals/landscapes/land1.jpg
        compressed/landscapes_opt/land1opt.jpg
    """
    original_path = os.path.join(originals_dir, category_name)
```

```
    compressed_opt_path = os.path.join(compressed_dir,
f"{category_name}_opt")

    original_files = glob.glob(os.path.join(original_path,
f"{prefix}*.jpg"))
    if not original_files:
        print(f"Warning: no originals for {category_name}")
        return pd.DataFrame()

    rows = []

    for og_file_path in original_files:
        base = os.path.basename(og_file_path)       # land1.jpg
        image_id = os.path.splitext(base)[0]        # land1

        og = cv2.imread(og_file_path)
        if og is None:
            print(f"Skipping {base}: cannot load original.")
            continue

        comp_name = f"{image_id}opt.jpg"            # land1opt.jpg
        comp_path = os.path.join(compressed_opt_path, comp_name)
        comp = cv2.imread(comp_path)

        if comp is None:
            print(f"Missing optimized file: {comp_name}")
            continue

        mse, ssim_score = calculate_metrics(og, comp)

        num_part = ''.join(filter(str.isdigit, image_id))
        num_id = int(num_part) if num_part else 0

        rows.append({
            "Image_ID": image_id + "opt",
            "Image_Num_ID": num_id,
            "MSE": round(mse, 4),
            "SSIM": round(ssim_score, 4)
        })

    df = pd.DataFrame(rows)
    if df.empty:
        return df

    df =
df.sort_values("Image_Num_ID").drop(columns=["Image_Num_ID"]).reset_in
dex(drop=True)
    return df
```

```python
# ----- RUN: KNN on LANDSCAPES -----
originals_dir = r"originals"
compressed_dir = r"compressed"

land_df = generate_opt_matrix(
    category_name="landscapes",
    prefix="land",
    originals_dir=originals_dir,
    compressed_dir=compressed_dir
)

print("--- OPTIMIZED MATRIX: LANDSCAPES (KNN) ---")
print(land_df.to_markdown(index=False))

print("\n--- LANDSCAPES (KNN) AVERAGES ---")
print(f"Average MSE  : {land_df['MSE'].mean():.4f}")
print(f"Average SSIM : {land_df['SSIM'].mean():.4f}")
```

```
--- OPTIMIZED MATRIX: LANDSCAPES (KNN) ---
| Image_ID    |      MSE |   SSIM |
|:------------|---------:|-------:|
| land1opt    |  60.5248 | 0.9521 |
| land2opt    |  65.0525 | 0.9598 |
| land3opt    |  78.3523 | 0.9104 |
| land4opt    |  53.1993 | 0.9733 |
| land5opt    |  93.3887 | 0.9217 |
| land6opt    |  73.6987 | 0.9656 |
| land7opt    |  59.6828 | 0.9444 |
| land8opt    |  97.7992 | 0.9581 |
| land9opt    |  71.4247 | 0.9419 |
| land10opt   |  59.6316 | 0.9582 |
| land11opt   |  66.2949 | 0.9594 |
| land12opt   |  48.1743 | 0.9804 |
| land13opt   |  28.6206 | 0.9662 |
| land14opt   |  79.9504 | 0.9325 |
| land15opt   |  77.6933 | 0.9296 |
| land16opt   |  71.9145 | 0.9408 |
| land17opt   | 129.935  | 0.9512 |
| land18opt   |  48.4453 | 0.9675 |
| land19opt   |  80.8786 | 0.9578 |
| land20opt   | 235.049  | 0.8934 |
| land21opt   |  57.6697 | 0.9378 |
| land22opt   |  58.8201 | 0.9659 |
| land23opt   |  67.1414 | 0.9778 |

--- LANDSCAPES (KNN) AVERAGES ---
Average MSE  : 76.6670
Average SSIM : 0.9498
```

Analysis

As evident by the compressed images, the colors are approximated using K-Means method and then stored into clusters based on their similarity. From this, an approximation of the colors is formed and applied to all the pixels that fall within that particular cluster. This means that the larger the defined K value is, the closer the produced image will be to the original image.

In terms of the error, portraits have the lowest average MSE at 276.53 and a low overall SSIM of 0.7857. This indicates an overall low compression quality for this category of image. While the numerical change between a pixel's color value is low (MSE), the low SSIM means that the structural information of the entire image is poor. A possible explanation for these values would be the smoothness of the subjects' skin, subtle lighting with tonal gradients, and blurred backgrounds. All of these factors contribute to making a complex image that is difficult for the colors to be approximated and put into clusters.

On the other hand, Still Life images have the highest average SSIM at 0.8464, and an average MSE of 305.19. This indicates that it has the best preservation of quality after the compression algorithm is run. An explanation for this high performance is from the nature of Still Life images as a whole, where objects are well defined and have relatively smooth surfaces, making color clustering an effective method to use for compression.

Applying the optimized K-means algorithm and K nearest neighbors to architecture and landscapes drastically improved the image quality. Average MSE for architecture and landscapes were 352.4702 and 76.6670 respectably. From the results, it appears that applying KNN brought upon it the best image compressed in terms of MSE and SSIM scores, however it took 2-3 minutes to process each image. The impact of optimizing the K-means algorithm by running it at K = 32 also increased the processing times to 2-3 minutes aswell. Overall, both algorithms are effective, however running these algorithms for the rest of our findings would be computationally expensive and time consuming.

Conclusion

Going back to our hypothesis and cross referencing it to our matrix of results for each category, we can conclude that compression performance is indeed worse for images with higher edge density and color diversity when the k value is low. Architecture and Urban images exhibit high average MSE at k=4 (452 and 453 respectively), and low SSIM (0.796 and 0.729 respectively). This combination indicates a low compression performance due to there being an insufficient amount of clusters to accomodate the high variety of colors, ultimately resulting in a major loss in quality after compression is applied. Thus, the hypothesis is reinforced.

Team Contribution

Ryan Kim:

- Image dataset collection and category contribution.
- File compression testing and exporting for various k values across all categories.

- Video report editing and documentation.

Stuart Arief:

- Analysis and Conclusion

Lydia Niu

- EDA of the error metrics summary of MSE and SSIM
- Final report compilation

Jonathan Chun -

- Optimized architecture through K-means
- Optimized landscape through KNN

Colin Pham:

- Code image compression system
- work on report