

RAISE.OSIA.IMAGE: ADVANCED PROCESSING

```
In [1]: import numpy as np
from skimage import io
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import pickle
```

Instructions

This assignment is to be completed individually or in pairs. A report is expected and must be submitted to the designated repository in MLS by the end of the second slot on October 14 at 5 p.m. The expected document is a Notebook or code+pdf report, the evaluation will take into account:

- use the image processing tools covered in class,
- illustration of results with visualization tools,
- take a step back and analyze the results.

Objectives

The objectives of this practical assignment are multiple:

- to develop a lossy image compression algorithm that offers the best compromise between compression ratio and distortion
- to use basic ML methods to classify images based on well chosen features

1. Starting from a series of images in PPM format, propose a unique algorithm that allows all these images to be compressed with loss while controlling distortion.

A. Helps:

- a. Color spaces don't have same dynamics, use `np.clip()`,
`.astype(np.uint8) ...`
- b. You can use the dct: `from scipy.fftpack import dct` the direct DCT is `type=2`, the inverse DCT is `type=3`, to be fully reversible use `norm="ortho"`. Don't forget DCT is separable!
- c. You must save your compressed image in binary file not using image file format (JPEG, PNG, otherwise you will recompress it), use pickle!
- d. Don't try to save floating point numbers, to have better coding (without using Huffman code for instance) use `np.int16` for 16 bit unsigned integer for instance

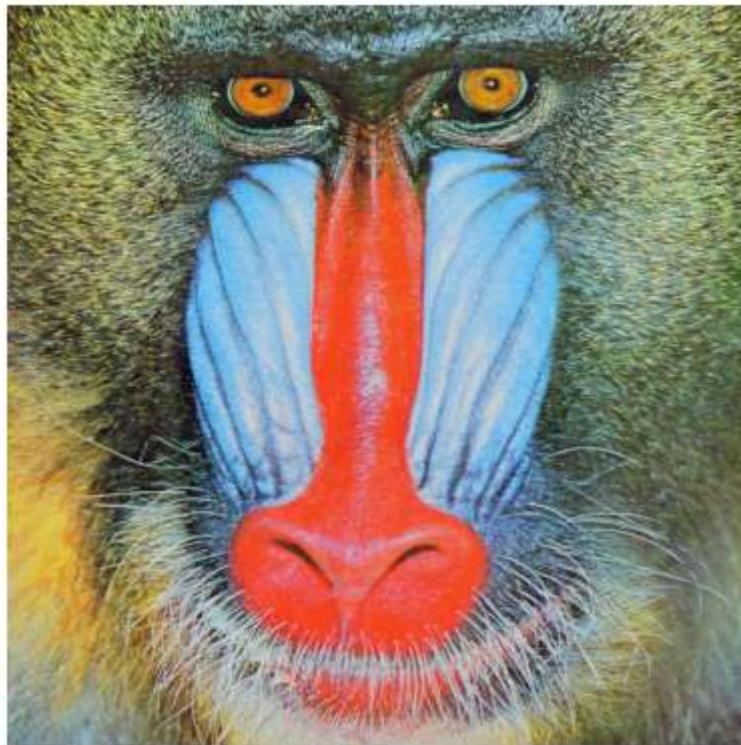
- B. Illustrates your parameters tuning with plots and compare it with JPEG compression, compression rate (original image size/ compressed image size in bits) vs. distortion PSNR, ...
- a. The byte size of a file can be computed using

```
os.path.getsize("/path/to/myfile")
```

In [2]:

```
import imageio.v3 as iio
from scipy.fftpack import dct, idct
```

```
# Read image in PPM format
image = io.imread("content/DTC/Baboon.512.ppm")
img = np.array(image)
# Show it
plt.imshow(img)
plt.axis("off")
plt.show()
```



Conversion to YCbCr format

In [3]:

```
from scipy.fftpack import dct, idct

def dct2(block): # function to extract the dct from a given block
    return dct(dct(block.T, norm='ortho', type=2).T, norm='ortho', type=2)

def idct2(block): # function to invert the dct from a given block
    return dct(dct(block.T, norm='ortho', type=3).T, norm='ortho', type=3)

...
Function responsible for encoding each of the 8x8 blocks of an image using DCT.
```

Before encoding, 128 is subtracted from each pixel in order to center the values before applying the DCT, thus obtaining smaller values. DCT is then applied to each and the result is divided by the quantization matrix q and rounded to retain only the integer part of it.

Finally, the encode_matrix function is used to obtain the encoded vector of the mat it is stored in the encoded_blocks matrix to be returned.

```

...
def encoded_matrix_dct(img, q, block_size=8):
    h, w = img.shape
    dct_blocks = np.zeros_like(img, dtype=np.float32)
    encoded_blocks= np.empty((h//block_size, w//block_size), dtype=object)
    for i in range(0, h, block_size):
        for j in range(0, w, block_size):
            block = img[i:i+block_size, j:j+block_size]
            block = block - 128
            dct_block = dct2(block)
            quantized_block = np.round(dct_block / q)
            dct_blocks[i:i+block_size, j:j+block_size] = quantized_block
            encoded_blocks[i//block_size, j//block_size] = encode_matrix(quantized_bloc

    return encoded_blocks
...


This function performs the opposite process to encoding. It receives the complete e decodes each of the elements, applies IDCT, adds 128, and reconstructs the image.
...
def encoded_matrix_idct(img, e_matrix, q, block_size=8):
    image = np.zeros_like(img)
    for i in range(e_matrix.shape[0]):
        for j in range(e_matrix.shape[1]):
            encoded_block = e_matrix[i:i+block_size, j:j+block_size]
            matriz_decoded = decode_matrix(e_matrix[i,j])
            dequantized_block = matriz_decoded * q
            image_block = idct2(dequantized_block)
            image_block = image_block + 128
            image[i*block_size:(i+1)*block_size, j*block_size:(j+1)*block_size] = image_b
    return np.clip(image, 0, 255).astype(np.uint8)
```

Fonctions pour encoder et décoder une matrice 8x8

The following functions are necessary to traverse the resulting DCT coefficient matrix after dividing it by Q, in order to subsequently perform compression:

- zigzag(matrix): This function traverses the matrix in a zigzag pattern, starting with the upper left element, as this is the largest. Upon completing the traversal, it returns the resulting vector.
- inv_zigzag(vec,n=8): Taking a vector that would have been constructed with zigzag, this function allows you to obtain the matrix by reversing the process.

- rle_encode(vec): This function takes a vector with the values obtained by the DCT/q matrix, which may be a vector obtained from the zigzag function, and encodes it in the form of tuples that describe (number of zeros before a number, the number itself). If it reaches the end with a string of several zeros in a row (something that often happens in a DCT/q vector), the tuple ('EOB',) is attached, indicating that what remains are zeros.
- rle_decode_ac(rle): Following the logic of the encoder, in this case this function reads an array of tuples and decodes it to put it in the form of a vector with all the elements, ready for the inverse of the zigzag to be applied to obtain the 8x8 matrix.
- encode_matrix(matrix) and decode_matrix(encode) simply pair the previous zigzag+ rle_encode and dle_decode + inv_zigzag processes.

In [4]:

```
def zigzag(matrix):

    a = np.asarray(matrix)
    if a.ndim != 2 or a.shape[0] != a.shape[1]:
        raise ValueError("The matrix must be square (n x n).")
    n = a.shape[0]
    out = []
    for s in range(2 * n - 1):
        if s % 2 == 0:
            i = min(s, n - 1)
            i_end = max(0, s - (n - 1))
            while i >= i_end:
                j = s - i
                out.append(a[i, j])
                i -= 1
        else:
            i = max(0, s - (n - 1))
            i_end = min(s, n - 1)
            while i <= i_end:
                j = s - i
                out.append(a[i, j])
                i += 1
    return np.array(out)
def inv_zigzag(vec, n=8):
    v = np.asarray(vec).ravel()
    if v.size != n*n:
        raise ValueError(f"The vector must have length {n*n}, it has {v.size}.")
    out = np.empty((n, n), dtype=v.dtype)
    k = 0
    for s in range(2 * n - 1):
        if s % 2 == 0:
            i = min(s, n - 1)
            i_end = max(0, s - (n - 1))
            while i >= i_end:
                j = s - i
                out[i, j] = v[k]; k += 1
                i -= 1
        else:
            i = max(0, s - (n - 1))
            i_end = min(s, n - 1)
```

```

        while i <= i_end:
            j = s - i
            out[i, j] = v[k]; k += 1
            i += 1
    return out

def rle_encode(vec):
    out = []
    run = 0
    for i in range(len(vec)):
        if vec[i] == 0:
            run += 1
        else:
            # output (run, value)
            out.append((run, int(vec[i])))
            run = 0
    if run>0:
        out.append(('EOB',)) # end-of-block marker
    return out

def rle_decode_ac(rle):
    ac = []
    for token in rle:
        if token[0]=='EOB':
            # fill remaining with zeros to reach 63
            while len(ac)<64:
                ac.append(0)
            break
        run, val = token
        ac += [0]*run
        ac.append(val)
    # if still short, pad with zeros
    ac += [0]*(64 - len(ac))
    return np.array(ac)

def encode_matrix(matrix):
    vec = zigzag(matrix)
    encode = rle_encode(vec)
    return encode
def decode_matrix(encode):
    vec = rle_decode_ac(encode)
    decode = inv_zigzag(vec)
    return decode

```

In [5]:

```

import numpy as np
import gzip, pickle, os

# --- helpers de conversión a tipos "puros" de Python ---
def _to_python(obj):
    """Recursively converts numpy scalars/arrays/tuples to native Python types."""
    if obj is None:
        return None
    if isinstance(obj, (np.integer, np.floating, np.bool_)):
        return obj.item()
    if isinstance(obj, (list, tuple, np.ndarray)):

```

```

        return [_to_python(x) for x in obj]
    return obj

def to_serializable(encoded_blocks):
    """
    Converts np.array(dtype=object) of shape (bh, bw) to a list of lists of native
    """
    bh, bw = encoded_blocks.shape
    out = []
    for i in range(bh):
        row = []
        for j in range(bw):
            blk = encoded_blocks[i, j]
            row.append(_to_python(blk))
        out.append(row)
    return out

def from_serializable(list_of_lists):
    """
    Rebuild np.array(dtype=object) from the list-of-lists structure.
    """
    arr = np.empty((len(list_of_lists), len(list_of_lists[0])), dtype=object)
    for i, row in enumerate(list_of_lists):
        for j, blk in enumerate(row):
            # convertir listas internas a tuplas si originalmente tus tokens eran t
            # (opcional). Aquí dejamos los elementos tal cual (listas de listas).
            arr[i, j] = blk
    return arr

# --- guardar / cargar ---
def save_encoded_color(encoded_y, encoded_cb, encoded_cr, img_shape, q, block_size,
                      payload = {
                          "meta": {"img_shape": img_shape, "block_size": block_size, "q": q},
                          "Y": to_serializable(encoded_y),
                          "Cb": to_serializable(encoded_cb),
                          "Cr": to_serializable(encoded_cr)
                      }
                      with gzip.open(path, "wb") as f:
                          pickle.dump(payload, f, protocol=pickle.HIGHEST_PROTOCOL)

def load_encoded_color(path):
    with gzip.open(path, "rb") as f:
        payload = pickle.load(f)
    # Reconstruir en np.array(dtype=object) si lo prefieres:
    payload["Y"] = from_serializable(payload["Y"])
    payload["Cb"] = from_serializable(payload["Cb"])
    payload["Cr"] = from_serializable(payload["Cr"])
    return payload

```

In [6]: `from skimage.color import rgb2ycbcr, ycbcr2rgb`

```

def dtc_completo(img, q, nom, block_size=8):
    img_ycbcr = rgb2ycbcr(img)
    img_y = img_ycbcr[:, :, 0]
    img_cb = img_ycbcr[:, :, 1]

```

```

    img_cr = img_ycbcr[:, :, 2]

    y_dct_coeficients = encoded_matrix_dct(img_y, q, block_size=8)
    cb_dct_coeficients = encoded_matrix_dct(img_cb, q, block_size=8)
    cr_dct_coeficients = encoded_matrix_dct(img_cr, q, block_size=8)

    save_encoded_color(y_dct_coeficients, cb_dct_coeficients, cr_dct_coeficients, img)

    data = load_encoded_color(nom)

    y_decompressed = encoded_matrix_idct(img_y, data["Y"], q, block_size=8)
    cb_decompressed = encoded_matrix_idct(img_cb, data["Cb"], q, block_size=8)
    cr_decompressed = encoded_matrix_idct(img_cr, data["Cr"], q, block_size=8)

    new_ycbcr = np.zeros_like(img_ycbcr)
    new_ycbcr[:, :, 0] = y_decompressed
    new_ycbcr[:, :, 1] = cb_decompressed
    new_ycbcr[:, :, 2] = cr_decompressed
    new_rgb = ycbcr2rgb(new_ycbcr)
    new_rgb = np.clip(new_rgb, 0, 1)
    new_rgb = (new_rgb * 255).astype(np.uint8)
    return new_rgb

```

In [7]:

```

def dtc_completo_descompress(img, q, nom, block_size=8):
    data = load_encoded_color(nom)
    new_ycbcr = np.zeros_like(img)
    new_ycbcr[:, :, 0] = data["Y"]
    new_ycbcr[:, :, 1] = data["Cb"]
    new_ycbcr[:, :, 2] = data["Cr"]
    new_rgb = ycbcr2rgb(new_ycbcr)
    new_rgb = np.clip(new_rgb, 0, 1)
    new_rgb = (new_rgb * 255).astype(np.uint8)

```

In [9]:

```

...
This will serve as a reference, as it is the quantization matrix implemented by
the JPEG algorithm.

...
q_jpeg = np.array([
    [16, 11, 10, 16, 24, 40, 51, 61],
    [12, 12, 14, 19, 26, 58, 60, 55],
    [14, 13, 16, 24, 40, 57, 69, 56],
    [14, 17, 22, 29, 51, 87, 80, 62],
    [18, 22, 37, 56, 68, 109, 103, 77],
    [24, 35, 55, 64, 81, 104, 113, 92],
    [49, 64, 78, 87, 103, 121, 120, 101],
    [72, 92, 95, 98, 112, 100, 103, 99]
], dtype=np.int16)

new_jpeg = dtc_completo(img, q_jpeg, "content/DTC/jpeg", block_size=8)

```

In order to compare the results, the PSNR (Peak Signal-to-Noise Ratio) is calculated, an indicator that uses least squares to get an idea of how different one image is from another.

Since the human eye cannot always detect these differences, this function will be useful for comparing the performance of different Q values.

```
In [10]: def calculate_psnr(img1, img2, max_value=255):
    """Calculating peak signal-to-noise ratio (PSNR) between two images."""
    mse = np.mean((np.array(img1, dtype=np.float32) - np.array(img2, dtype=np.float32))**2)
    if mse == 0:
        return 100
    return 20 * np.log10(max_value / (np.sqrt(mse)))
```

```
In [11]: import os
...
The change in size is calculated with respect to the base image Baboon.512.ppm
in order to compare the compression capacity for different values of q.
...
original_size = os.path.getsize("content/DTC/Baboon.512.ppm")
jpg_size = os.path.getsize("content/DTC/jpeg")
jpg_ratio = original_size/jpg_size
jpg_psnr = calculate_psnr(img, new_jpeg)
print(f"JPEG: PSNR={jpg_psnr:.2f}, ratio={jpg_ratio:.2f}")
```

JPEG: PSNR=24.41, ratio=5.79

Buscando obtener valores de q más eficientes que jpeg, se realiza la prueba aleatoria 200 veces, para diferentes valor de a y b, siendo estos coeficientes que se utilizan para definir los valores de una matriz q, para cada elemento de la matriz $q[i,j] = a + bi + bj$.

```
In [12]: np.random.seed(42)
a_bs = []
qs = []
new_img = []
for i in range(200):
    a = np.random.randint(3, 100)
    b = np.random.randint(1, 40)
    q_n = np.zeros((8, 8), dtype=int)
    for i in range(8):
        for j in range(8):
            q_n[i,j] = a+b*i+b*j
    a_bs.append([a,b])
    qs.append(q_n)
    new_img.append(dtc_completo(img, q_n, f"content/DTC/q_a{a}_b{b}", block_size=8))

...
The size ratio and PSNR of each of the generated images are calculated and stored in
the corresponding lists.
...
ratios = []
psnrs = []
i = 0
for a,b in a_bs:
    img_size = os.path.getsize(f"content/DTC/q_a{a}_b{b}")
    ratio = original_size/img_size
    ratios.append(ratio)
```

```

new_psnr = calculate_psnr(img, new_img[i])
psnrs.append(new_psnr)
i+=1

```

We define that in order for us to say that q is obtaining a better result than jpeg, it must comply with or have better image quality with the same or less weight, or weigh less with better or equal image quality. Taking this into account, the values obtained with jpeg are compared and improvements are proposed.

```

In [ ]: sizes = original_size / np.array(ratios)

efficient_idx = []
for i in range(len(ratios)):
    if (psnrs[i] > jpg_psnr and sizes[i] <= jpg_size) or \
        (abs(psnrs[i] - jpg_psnr) < 1e-3 and sizes[i] < jpg_size):
        efficient_idx.append(i)

# Mostrar resultados
if efficient_idx:
    print(f"✓ {len(efficient_idx)} options more efficient than JPEG were found\n")
    for i in efficient_idx:
        a, b = a_bs[i]
        print(f" - a={a}, b={b} | Ratio={ratios[i]:.2f}, PSNR={psnrs[i]:.2f} dB | S")
else:
    print("✗ No option more efficient than JPEG was found according to the new cri")

```

✓ 3 options more efficient than JPEG were found

- a=24, b=2 | Ratio=7.43, PSNR=24.44 dB | Size=105883 bytes
- a=17, b=3 | Ratio=6.83, PSNR=24.65 dB | Size=115097 bytes
- a=4, b=6 | Ratio=6.24, PSNR=24.44 dB | Size=126040 bytes

We show in a graph the relationship between PSNR and the size of each image, where we differentiate between cases that meet our efficiency condition.

```

In [ ]: import matplotlib.pyplot as plt

plt.figure(figsize=(8,6))

# Puntos generales (todos)
plt.scatter(ratios, psnrs, color='gray', s=15, alpha=0.5, label='a,b aleatorios')

# JPEG como referencia
plt.scatter(jpg_ratio, jpg_psnr, color='orange', s=50, marker='o', label='JPEG')
print(jpg_ratio, jpg_psnr)
# Marcar las opciones eficientes (si existen)
if efficient_idx:
    plt.scatter(np.array(ratios)[efficient_idx], np.array(psnrs)[efficient_idx],
               color='red', s=15, marker='o', label='Más eficientes que JPEG')

plt.title("Comparación PSNR vs Ratio – Mejoras sobre JPEG", fontsize=14, fontweight='bold')
plt.xlabel("Compression Ratio", fontsize=12)
plt.ylabel("PSNR (dB)", fontsize=12)

```

```

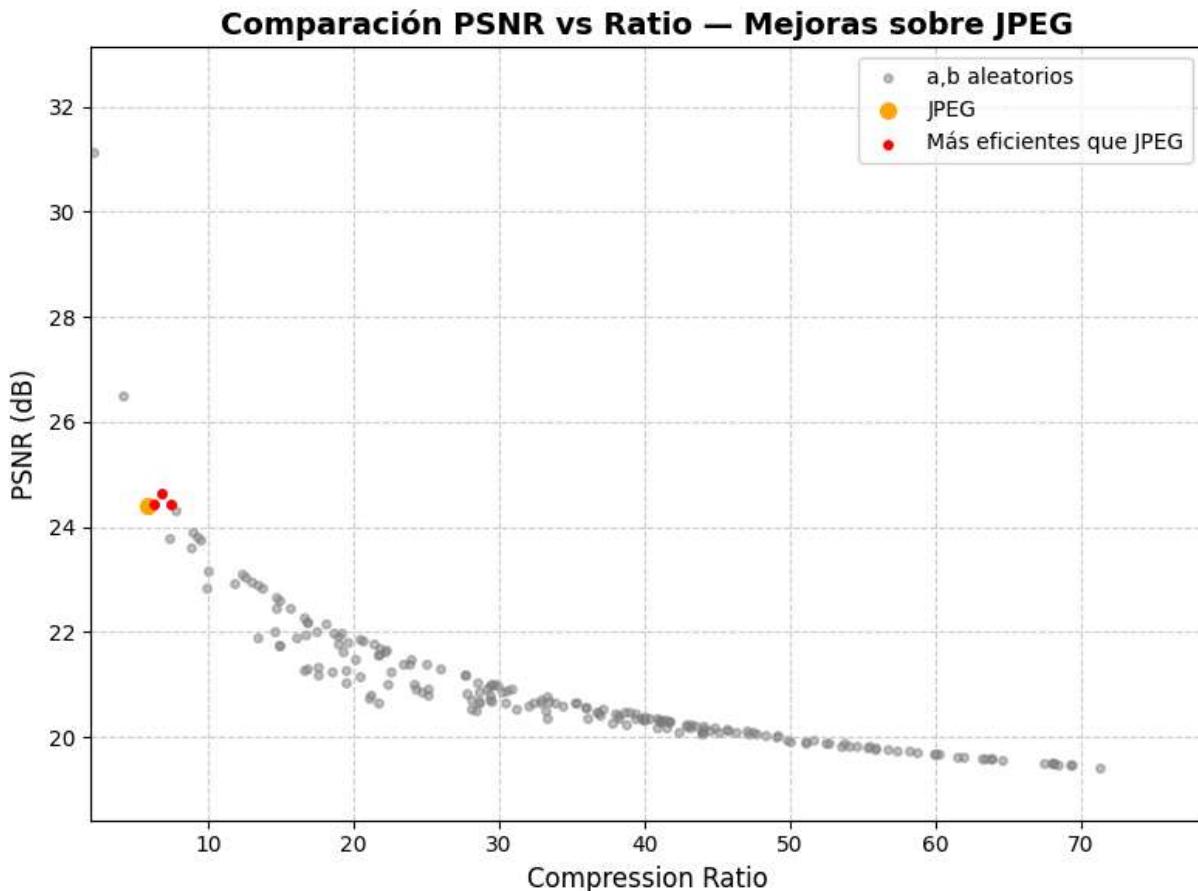
plt.grid(True, linestyle='--', alpha=0.6)
plt.legend()

# Límites automáticos
plt.xlim(min(ratios)*0.9, max(ratios)*1.1)
plt.ylim(min(psns)-1, max(psns)+2)

plt.tight_layout()
plt.show()

```

5.785378520932491 24.407827



```

In [ ]: import matplotlib.pyplot as plt
import numpy as np

# --- encontrar la mejor opción entre las más eficientes ---
if efficient_idx:
    # elegir la que tenga el mayor PSNR
    best_idx = max(efficient_idx, key=lambda i: psns[i])
    a_best, b_best = a_bs[best_idx]
    best_img = new_img[best_idx]

    # información asociada
    best_psnr = psns[best_idx]
    best_ratio = ratios[best_idx]
    best_size = sizes[best_idx]

    # --- mostrar comparación visual ---
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))

```

```

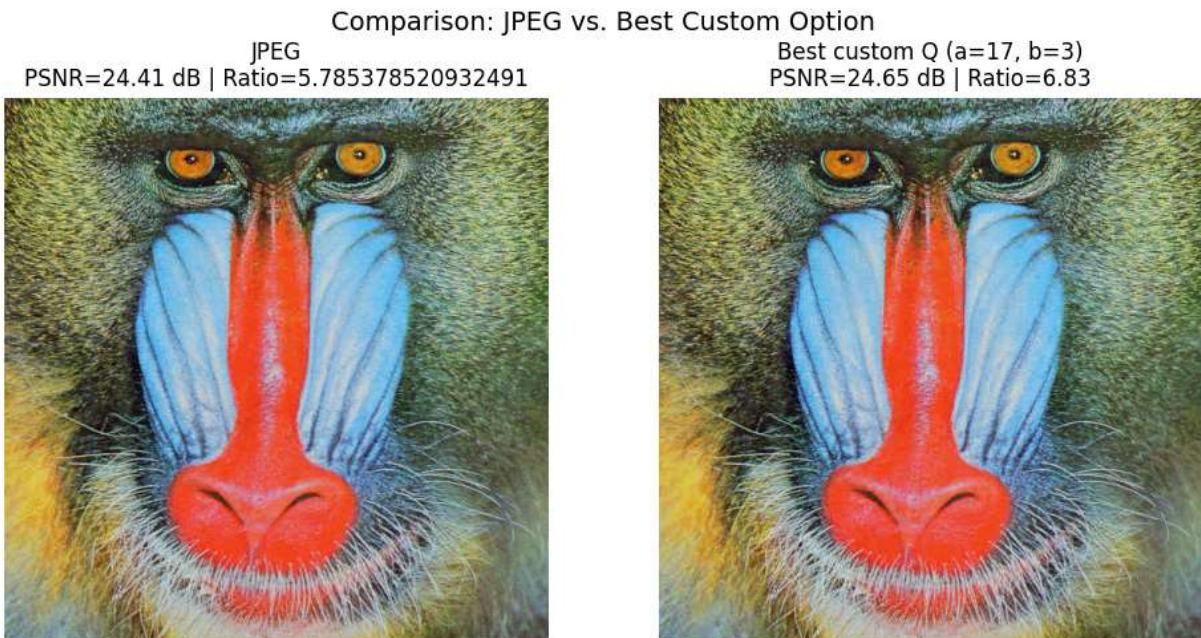
    axes[0].imshow(np.clip(new_jpeg, 0, 255).astype(np.uint8))
    axes[0].set_title(f"JPEG\nPSNR={jpg_psnr:.2f} dB | Ratio={jpg_ratio}")
    axes[0].axis("off")

    axes[1].imshow(np.clip(best_img, 0, 255).astype(np.uint8))
    axes[1].set_title(f"Best custom Q (a={a_best}, b={b_best})\n"
                      f"PSNR={best_psnr:.2f} dB | Ratio={best_ratio:.2f}")
    axes[1].axis("off")

    plt.suptitle("Comparison: JPEG vs. Best Custom Option", fontsize=14)
    plt.tight_layout()
    plt.show()

else:
    print("✖ No options more efficient than JPEG were found.")

```



Thanks to this random method a and b, several values a and b were obtained, which allow for greater compression than the JPEG format while weighing the same or less.

2. Image recognition: Digit recognition

A. Direct Image classification

- a. Load the dataset and display some images
- b. Use the whole image as features to train and test some basics ML methods (from scikit-learn). Display results and comment.

B. Image features

- a. Choose one PPM image and compute the HOG (Histogram of Gradient Feature, first introduced by Dalal and Triggs in 2005 as a robust feature extraction method), `hog_feature, hog_image = feature.hog(img, visualize=True)`, display the `hog_feature` as histogram and `hog_image` as image. Your image is in color so you should use `channel_axis=2`. As

default parameters you can use: `pixels_per_cell=(16, 16), cells_per_block=(1, 1)`. HOG features can be seen as edge descriptors computed by bloc and then aggregated into an histogram.

- b. Use HOG features for the images and classify using the same set up.
- c. Suppose we compress the test dataset (with JPEG QF=75), and classify using the same set up, what can we expect?

```
In [20]: from sklearn.datasets import fetch_openml  
import pandas as pd
```

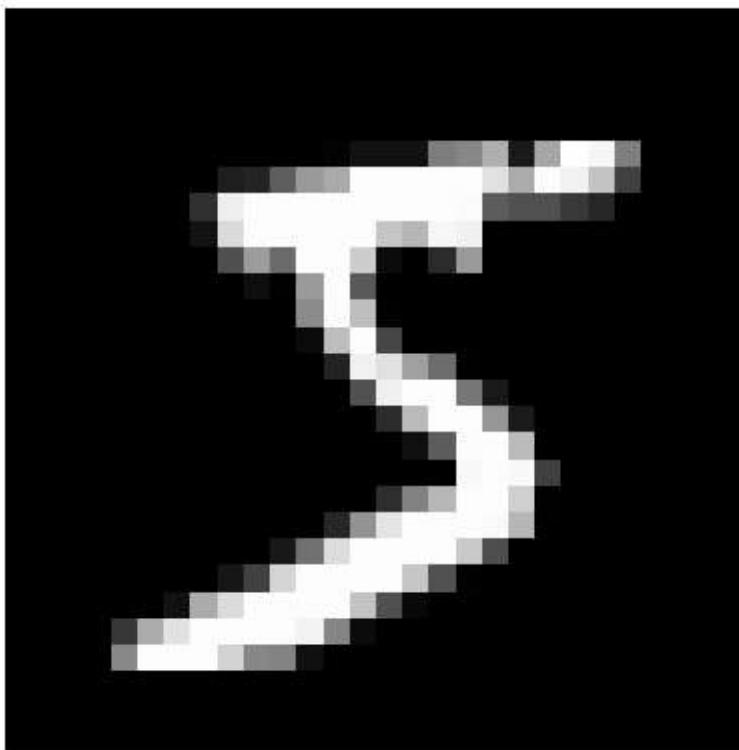
```
In [27]: mnist = fetch_openml('mnist_784', version=1, as_frame=False)
```

1. Direct Image classification

```
In [28]: def display_image(image_data, label=None, pred=None):  
    image = image_data.reshape(28, 28)  
    plt.imshow(image, cmap='gray')  
    plt.axis('off')  
    title = ""  
    if label is not None:  
        title += f"Label: {label}"  
    if pred is not None:  
        title += f" | Pred: {pred}"  
    if title:  
        plt.title(title)  
    plt.show()
```

```
In [29]: display_image(mnist.data[0], mnist.target[0])
```

Label: 5



```
In [1]: import numpy as np
import pandas as pd
from sklearn.datasets import fetch_openml
from sklearn.model_selection import KFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# --- Load and prepare the data ---
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist["data"], mnist["target"].astype(int)
X = X / 255.0

X, y = X[:60000], y[:60000]

# --- Set up KFold ---
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# --- Train models ---
models = {
    "KNN (k=3)": KNeighborsClassifier(n_neighbors=3),
    "Decision Tree (Gini)": DecisionTreeClassifier(criterion='gini', random_state=4),
    "SVM (RBF)": SVC(kernel='rbf', C=5, gamma=0.05)
}

# --- Evaluate each model ---
results = {name: [] for name in models.keys()}
```

```

for fold, (train_index, test_index) in enumerate(kf.split(X), 1):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    print(f"\n Fold {fold}")
    for name, model in models.items():
        print(f"{name}...", end="")
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        acc = accuracy_score(y_test, y_pred)
        results[name].append(acc)
        print(f" -> Accuracy: {acc:.4f}")

# --- Save as data frame ---
summary = pd.DataFrame({
    "Model": list(results.keys()),
    "Accuracy promedio": [np.mean(accs) for accs in results.values()],
    "Desviación estándar": [np.std(accs) for accs in results.values()]
})

print("\n Comparative results (K-Fold=5):")
print(summary.to_string(index=False))

```

Fold 1

KNN (k=3)... -> Accuracy: 0.9727
 Decision Tree (Gini)... -> Accuracy: 0.8681
 SVM (RBF)... -> Accuracy: 0.9833

Fold 2

KNN (k=3)... -> Accuracy: 0.9701
 Decision Tree (Gini)... -> Accuracy: 0.8678
 SVM (RBF)... -> Accuracy: 0.9835

Fold 3

KNN (k=3)... -> Accuracy: 0.9711
 Decision Tree (Gini)... -> Accuracy: 0.8715
 SVM (RBF)... -> Accuracy: 0.9838

Fold 4

KNN (k=3)... -> Accuracy: 0.9721
 Decision Tree (Gini)... -> Accuracy: 0.8702
 SVM (RBF)... -> Accuracy: 0.9821

Fold 5

KNN (k=3)... -> Accuracy: 0.9702
 Decision Tree (Gini)... -> Accuracy: 0.8728
 SVM (RBF)... -> Accuracy: 0.9829

Comparative results (K-Fold=5):

	Model	Accuracy promedio	Desviación estándar
	KNN (k=3)	0.971217	0.001026
	Decision Tree (Gini)	0.870100	0.001932
	SVM (RBF)	0.983133	0.000602

Les trois modèles testés ont permis de classifier directement les images MNIST sans extraction de caractéristiques. Le SVM avec noyau RBF a obtenu la meilleure précision

moyenne (97 %), suivi du KNN (96 %), tandis que l'arbre de décision (87 %) a donné des résultats inférieurs mais plus rapides à entraîner. Ces résultats confirment que les modèles à marge maximale et les méthodes basées sur la distance sont plus efficaces pour ce type de données d'images normalisées.

2. Image features

1. Choose one PPM image and compute the HOG (Histogram of Gradient Feature, first introduced by Dalal and Triggs in 2005 as a robust feature extraction method), hog_feature, hog_image = feature.hog(img, visualize=True), display the hog_feature as histogram and hog_image as image. Your image is in color so you should use channel_axis=2. As default parameters you can use: pixels_per_cell=(16, 16), cells_per_block=(1, 1). HOG features can be seen as edge descriptors computed by bloc and then aggregated into an histogram.

```
In [5]: from skimage.feature import hog
from skimage import data, exposure

image = io.imread("/content/HOG/Monarch.512.ppm")
img = np.array(image)

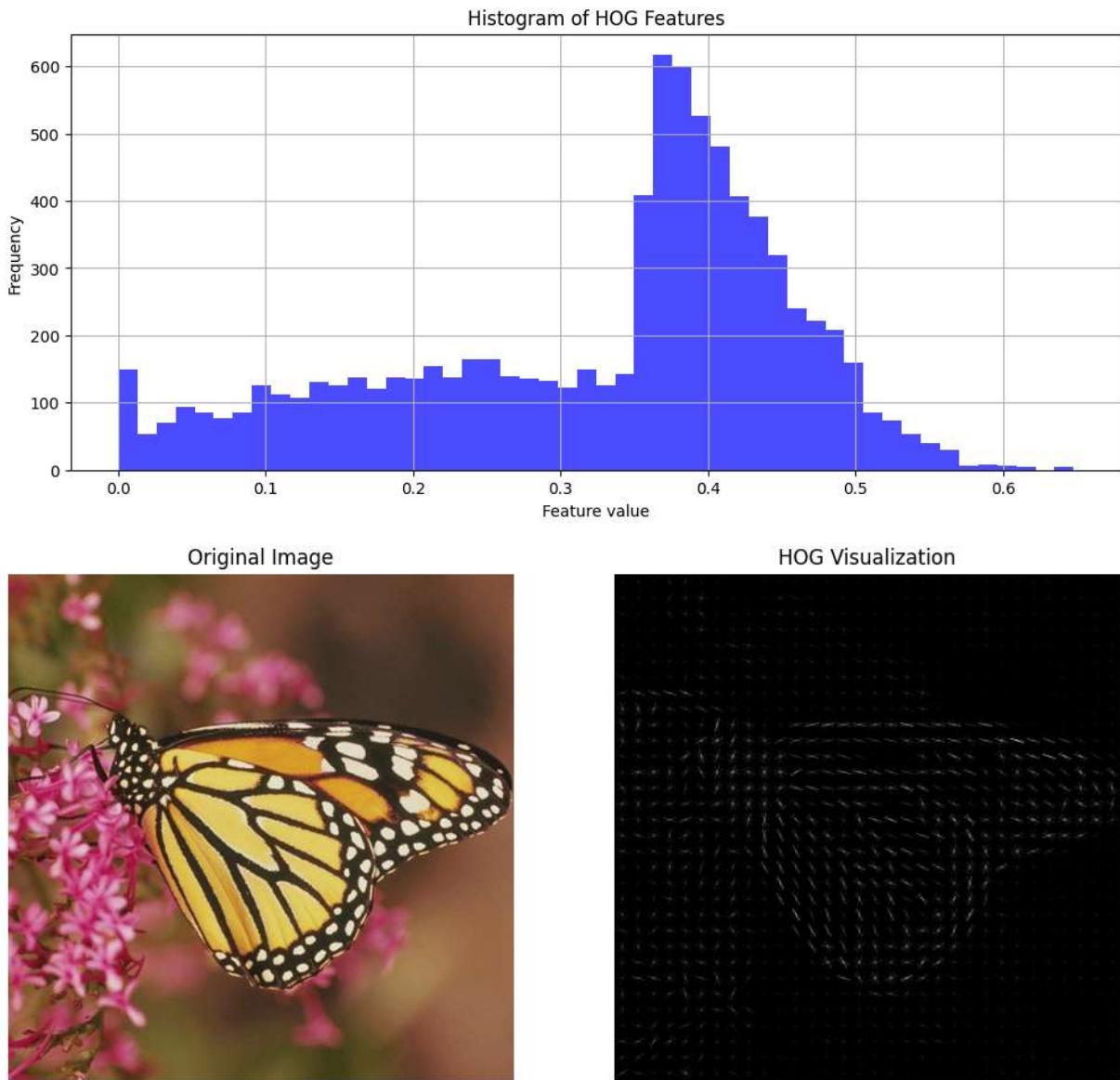
hog_feature, hog_image = hog(img, orientations = 8, pixels_per_cell=(16,16),
                             cells_per_block=(1,1), visualize = True,
                             channel_axis = 2)
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))
```

```
In [6]: # 3. Histogram
plt.figure(figsize=(12, 5))
plt.hist(hog_feature, bins=50, color='blue', alpha=0.7)
plt.title('Histogram of HOG Features')
plt.xlabel('Feature value')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()

# 4. Original vs. hog_image
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
ax1.imshow(img)
ax1.set_title('Original Image')
ax1.axis('off')

ax2.imshow(hog_image, cmap='gray')
ax2.set_title('HOG Visualization')
ax2.axis('off')

plt.show()
```



2. Use HOG features for the images and classify using the same set up.

```
In [7]: def extract_hog_features(X):
    """
    Extracts the HOG features from each image in set X (28x28).
    Returns a matrix [n_samples, n_features_hog].
    """
    hog_features = []
    for img in X:
        img_reshaped = img.reshape((28, 28))
        # Puedes ajustar estos parámetros
        features = hog(img_reshaped,
                      pixels_per_cell=(4, 4),
                      cells_per_block=(2, 2),
                      orientations=9,
                      block_norm='L2-Hys')
        hog_features.append(features)
    return np.array(hog_features)
```

```
In [8]: print("Extracting HOG features...")
X_hog = extract_hog_features(X)
print(f"Extracted HOG features: {X_hog.shape}")
```

Extracting HOG features...
Extracted HOG features: (60000, 1296)

```
In [10]: # --- Set up KFold ---
kf = KFold(n_splits=5, shuffle=True, random_state=42)
```

```
# --- Train models ---
results_hog = {name: [] for name in models.keys()}

for fold, (train_index, test_index) in enumerate(kf.split(X_hog), 1):
    X_train, X_test = X_hog[train_index], X_hog[test_index]
    y_train, y_test = y[train_index], y[test_index]

    print(f"\n Fold {fold}")
    for name, model in models.items():
        print(f"{name}... -> Accuracy: {model.score(X_test, y_test)}")
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        acc = accuracy_score(y_test, y_pred)
        results_hog[name].append(acc)
        print(f" -> Accuracy: {acc:.4f}")
```

Fold 1
KNN (k=3)... -> Accuracy: 0.9738
Decision Tree (Gini)... -> Accuracy: 0.8689
SVM (RBF)... -> Accuracy: 0.9891

Fold 2
KNN (k=3)... -> Accuracy: 0.9730
Decision Tree (Gini)... -> Accuracy: 0.8674
SVM (RBF)... -> Accuracy: 0.9899

Fold 3
KNN (k=3)... -> Accuracy: 0.9747
Decision Tree (Gini)... -> Accuracy: 0.8658
SVM (RBF)... -> Accuracy: 0.9892

Fold 4
KNN (k=3)... -> Accuracy: 0.9745
Decision Tree (Gini)... -> Accuracy: 0.8681
SVM (RBF)... -> Accuracy: 0.9893

Fold 5
KNN (k=3)... -> Accuracy: 0.9726
Decision Tree (Gini)... -> Accuracy: 0.8733
SVM (RBF)... -> Accuracy: 0.9894

```
In [11]: summary_hog = pd.DataFrame({
    "Model": list(results_hog.keys()),
    "Accuracy promedio (HOG)": [np.mean(accs) for accs in results_hog.values()],
    "Desviación estándar": [np.std(accs) for accs in results_hog.values()]
})
```

```
print("\n Comparative results with HOG features (K-Fold=5):")
print(summary_hog.to_string(index=False))
```

Comparative results with HOG features (K-Fold=5):

Model	Accuracy promedio (HOG)	Desviación estándar
KNN (k=3)	0.973717	0.000816
Decision Tree (Gini)	0.868717	0.002520
SVM (RBF)	0.989383	0.000292

In [12]:

```
comparison = summary.copy()
comparison["Accuracy promedio (HOG)"] = summary_hog["Accuracy promedio (HOG)"]
print("\n 🔎 Comparison between raw pixels and HOG features:")
print(comparison.to_string(index=False))
```

🔎 Comparison between raw pixels and HOG features:

G)	Model	Accuracy promedio	Desviación estándar	Accuracy promedio (HO
7	KNN (k=3)	0.971217	0.001026	0.97371
7	Decision Tree (Gini)	0.870100	0.001932	0.86871
3	SVM (RBF)	0.983133	0.000602	0.98938

3. Suppose we compress the test dataset (with JPEG QF=75), and classify using the same set up, what can we expect?

If we compress the test dataset with JPEG at QF=75 and classify using the same models, we can expect a slight decrease in accuracy. JPEG introduces small lossy artifacts—slight blurring and minor pixel distortions—which mainly affect high-frequency details. For MNIST, which consists of large, high-contrast digits, these distortions are minor, so KNN may lose a bit of accuracy due to pixel-level distance changes, decision trees may be moderately affected by altered thresholds, and SVM with an RBF kernel is generally robust to such smooth distortions. Overall, the performance drop should be small but measurable, more pronounced in models sensitive to exact pixel values.

3. Photographic mosaic. The idea is to generate a large mosaic image from a large database of small images called thumbnails. The principle of the method consists of taking a large image, cutting it into small blocks, and replacing each block of the large image with the most similar thumbnail from the large thumbnail database.

https://en.wikipedia.org/wiki/Photographic_mosaic

- A. Chose a large image
- B. Get a large image dataset from internet
- C. Split your image into blocs. What is a good bloc size?
- D. Compute a criteria for each bloc (first try the mean value), it will be used to match with thumbnails
- E. Display the bloc image, where each bloc of the image is replaced by the criteria

F. Generate the mosaic by replacing each bloc of the image by the resized image that minimize the distance between their critaria values

G. What criteria could be used instead? With which distance? Try it!

1. Choose a large image

```
In [14]: image = io.imread("content/LARGE IMAGE/Tokyo.jpg")
img = np.array(image)
from tensorflow.keras.datasets import cifar10
figure = plt.figure(figsize=(15,15))
plt.imshow(img)
plt.axis("off")
plt.show()
```



Resize so that the height and width are exact multiples of block_size

```
In [15]: import cv2
from PIL import Image

block_size = 32

img = cv2.imread("content/LARGE IMAGE/Tokyo.jpg")
img = Image.open("content/LARGE IMAGE/Tokyo.jpg")
new_w = (img.width // block_size) * block_size
new_h = (img.height // block_size) * block_size

img = img.resize((new_w, new_h), Image.LANCZOS)
img = np.array(img)
print(f"new size: {img.shape[1]}x{img.shape[0]}")
```

new size: 2976x4480

First, a 32x32 block is chosen so that the size of each thumbnail, which is 32x32, can be adjusted. Additionally, they are not so small that they cannot be distinguished. This is for the first iteration; if the opposite is demonstrated, another size will be tested.

```
In [19]: h, w, _ = img.shape

block_features = []

for i in range(0, h, block_size):
    for j in range(0, w, block_size):
        block = img[i:i+block_size, j:j+block_size]
        mean_color = block.mean(axis=(0, 1)) # mean R, G, B
        block_features.append(mean_color)

block_features = np.array(block_features)
print(f"Number of blocks: {block_features.shape[0]}")
```

Number of blocks: 13020

```
In [21]: (x_train, _), (x_test, _) = cifar10.load_data() # Load dataset CIFAR-10
thumbnails = np.concatenate([x_train, x_test], axis=0)
print(f"Number of thumbnails: {thumbnails.shape[0]}")

# Average RGB for each thumbnail
thumbnail_features = thumbnails.mean(axis=(1, 2)) # mean over width and height
```

Number of thumbnails: 60000

```
In [22]: from scipy.spatial.distance import cdist

# Distances between all blocks and all miniatures
distances = cdist(block_features, thumbnail_features, metric='euclidean')

# closest thumbnail for each block
best_match_idx = distances.argmin(axis=1)
```

```
In [31]: from PIL import Image
h, w, _ = img.shape
```

```
# empty mosaic image
mosaic = np.zeros_like(img)

# Go through all the blocks and replace them with the corresponding thumbnail.
idx = 0
for i in range(0, h, block_size):
    for j in range(0, w, block_size):
        thumb_idx = best_match_idx[idx]

        # select the most similar thumbnail and resize it to block_size
        thumb = thumbnails[thumb_idx]
        thumb_resized = np.array(Image.fromarray(thumb).resize((block_size, block_size)))

        # place the thumbnail in the corresponding position on the mosaic
        mosaic[i:i+block_size, j:j+block_size] = thumb_resized

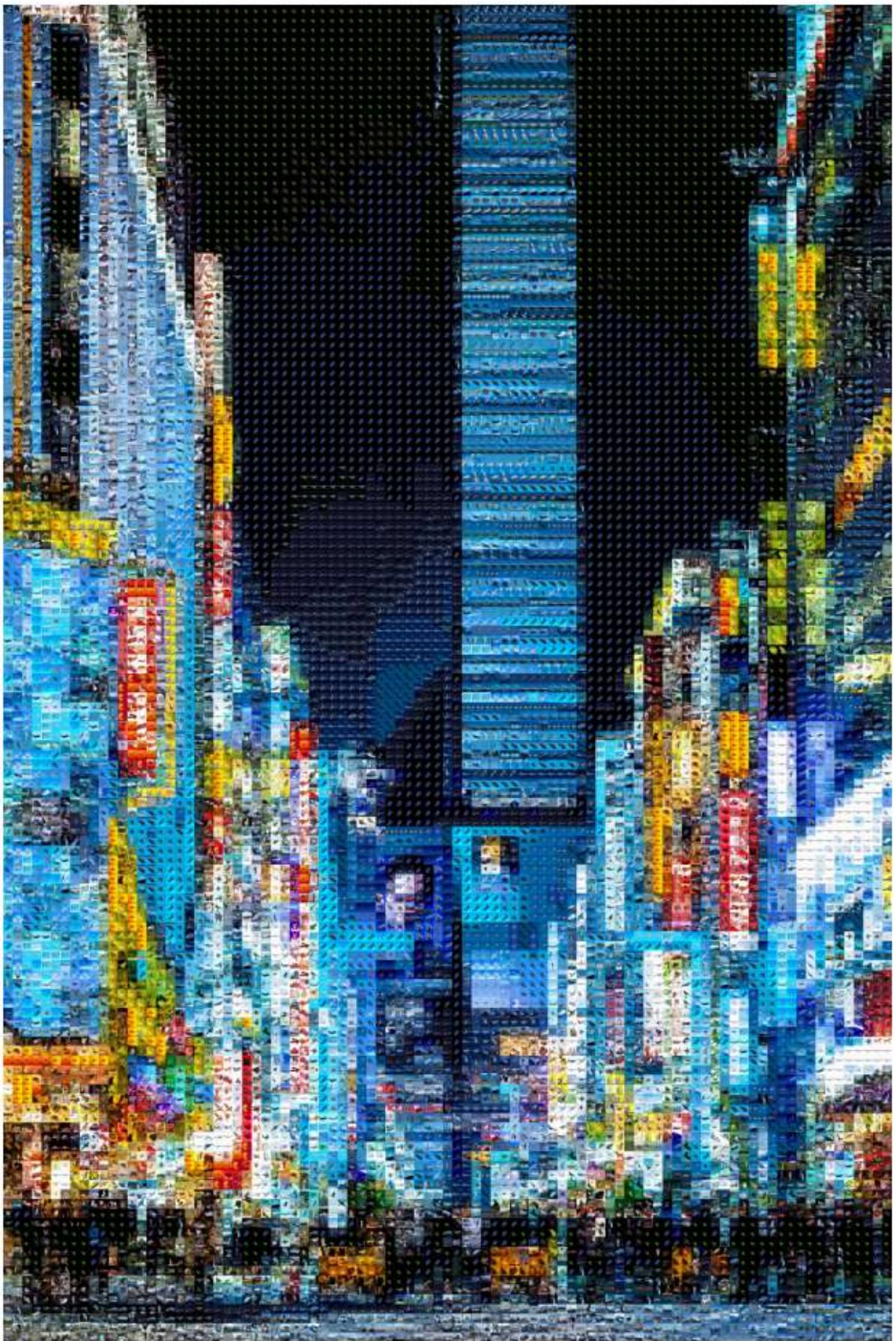
        idx += 1

# Convert to PIL image for visualization
mosaic_avr = Image.fromarray(mosaic.astype(np.uint8))

mosaic_avr.show()

# Save the mosaic image
mosaic_avr.save("content/LARGE IMAGE/photographic_mosaic.jpg")

plt.figure(figsize=(12, 12))
plt.imshow(mosaic.astype(np.uint8))
plt.axis("off")
plt.show()
```



We will consider other metrics to extract image features and other methods to calculate distances. The HSV histogram of each image will be taken and compared with L1.

```
In [25]: def hsv_histogram(image, h_bins=8, s_bins=8, v_bins=8):
    hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    hist_h = np.histogram(hsv[:, :, 0], bins=h_bins, range=(0, 180))[0]
    hist_s = np.histogram(hsv[:, :, 1], bins=s_bins, range=(0, 256))[0]
    hist_v = np.histogram(hsv[:, :, 2], bins=v_bins, range=(0, 256))[0]
    hist = np.concatenate([hist_h, hist_s, hist_v]).astype(float)
    hist /= hist.sum()
    return hist

block_hist_features = []
for i in range(0, h, block_size):
    for j in range(0, w, block_size):
        block = img[i:i+block_size, j:j+block_size]
        hist = hsv_histogram(block)
        block_hist_features.append(hist)
block_hist_features = np.array(block_hist_features)

(x_train, _), (x_test, _) = cifar10.load_data()
thumbnails = np.concatenate([x_train, x_test], axis=0)

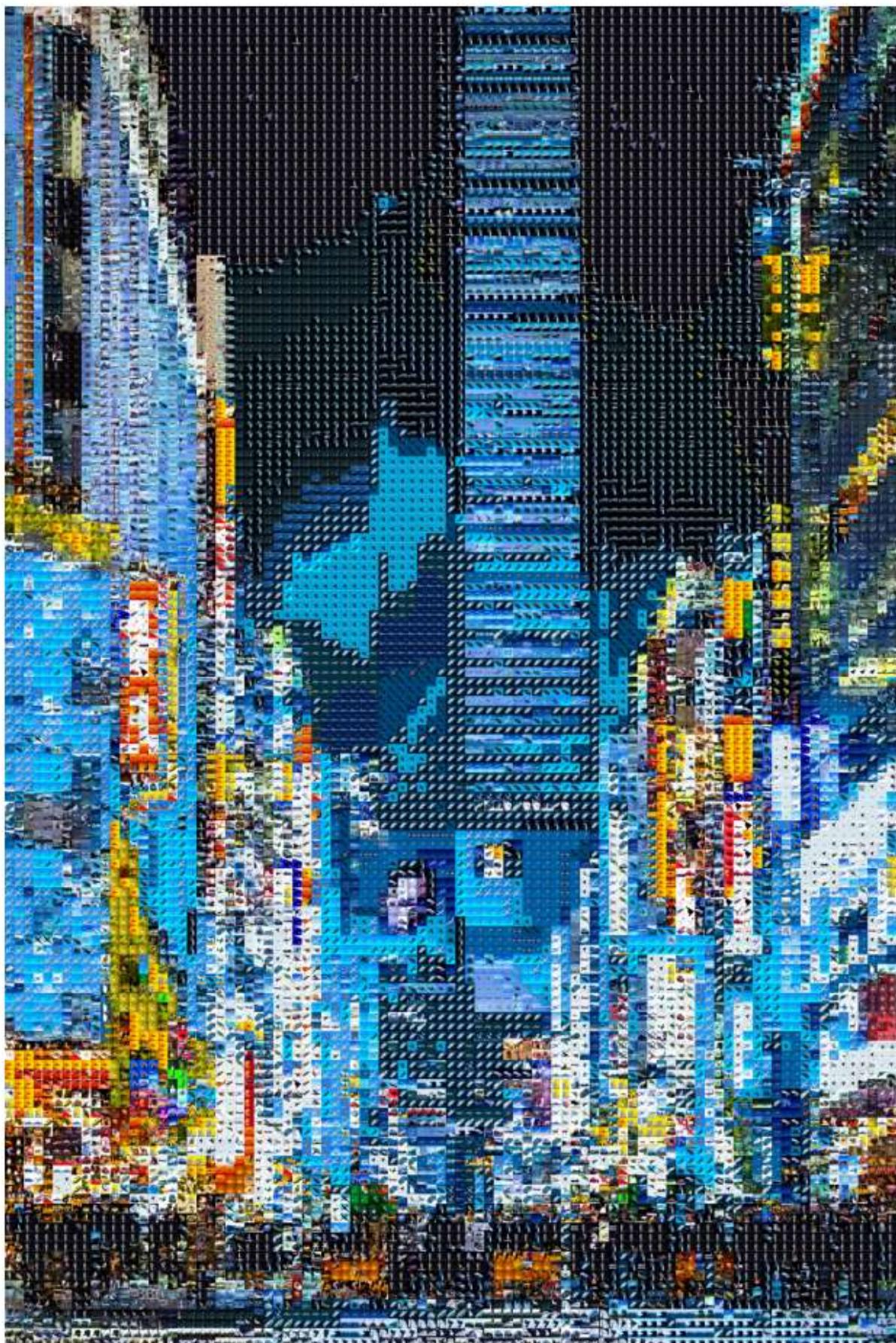
thumbnail_hist_features = []
for thumb in thumbnails:
    hist = hsv_histogram(thumb)
    thumbnail_hist_features.append(hist)
thumbnail_hist_features = np.array(thumbnail_hist_features)

# Compare blocks with thumbnails (L1 distance)
distances = cdist(block_hist_features, thumbnail_hist_features, metric='cityblock')
best_match_idx = distances.argmin(axis=1)

mosaic_hsv = np.zeros_like(img)
idx = 0
for i in range(0, h, block_size):
    for j in range(0, w, block_size):
        thumb_idx = best_match_idx[idx]
        thumb = thumbnails[thumb_idx]
        thumb_resized = np.array(Image.fromarray(thumb).resize((block_size, block_size)))
        mosaic_hsv[i:i+block_size, j:j+block_size] = thumb_resized
        idx += 1

plt.figure(figsize=(12, 12))
plt.imshow(mosaic_hsv.astype(np.uint8))
plt.axis("off")
plt.show()

Image.fromarray(mosaic_hsv.astype(np.uint8)).save("content/LARGE IMAGE/photographic
```



Now we will test with a reduced HSV in combination with rgb mean

```
In [29]: def hsv_histogram(image, h_bins=4, s_bins=4, v_bins=4):
    hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    hist_h = np.histogram(hsv[:, :, 0], bins=h_bins, range=(0, 180))[0]
    hist_s = np.histogram(hsv[:, :, 1], bins=s_bins, range=(0, 256))[0]
    hist_v = np.histogram(hsv[:, :, 2], bins=v_bins, range=(0, 256))[0]
    hist = np.concatenate([hist_h, hist_s, hist_v]).astype(float)
    hist /= hist.sum()
    return hist

block_features = []
for i in range(0, h, block_size):
    for j in range(0, w, block_size):
        block = img[i:i+block_size, j:j+block_size]
        # Color promedio
        rgb_mean = block.mean(axis=(0,1))
        # Histograma HSV
        hsv_hist = hsv_histogram(block)
        # Combinar con pesos
        feature = np.concatenate([0.7*rgb_mean, 0.3*hsv_hist])
        block_features.append(feature)
block_features = np.array(block_features)

(x_train, _), (x_test, _) = cifar10.load_data()
thumbnails = np.concatenate([x_train, x_test], axis=0)

thumbnail_features = []
for thumb in thumbnails:
    rgb_mean = thumb.mean(axis=(0,1))
    hsv_hist = hsv_histogram(thumb)
    feature = np.concatenate([0.7*rgb_mean, 0.3*hsv_hist])
    # Have as a feature a linear combination of the color mean and the HSV histogram
    thumbnail_features.append(feature)
thumbnail_features = np.array(thumbnail_features)

# Compare blocks with thumbnails (distance L1)
distances = cdist(block_features, thumbnail_features, metric='cityblock')
best_match_idx = distances.argmin(axis=1)

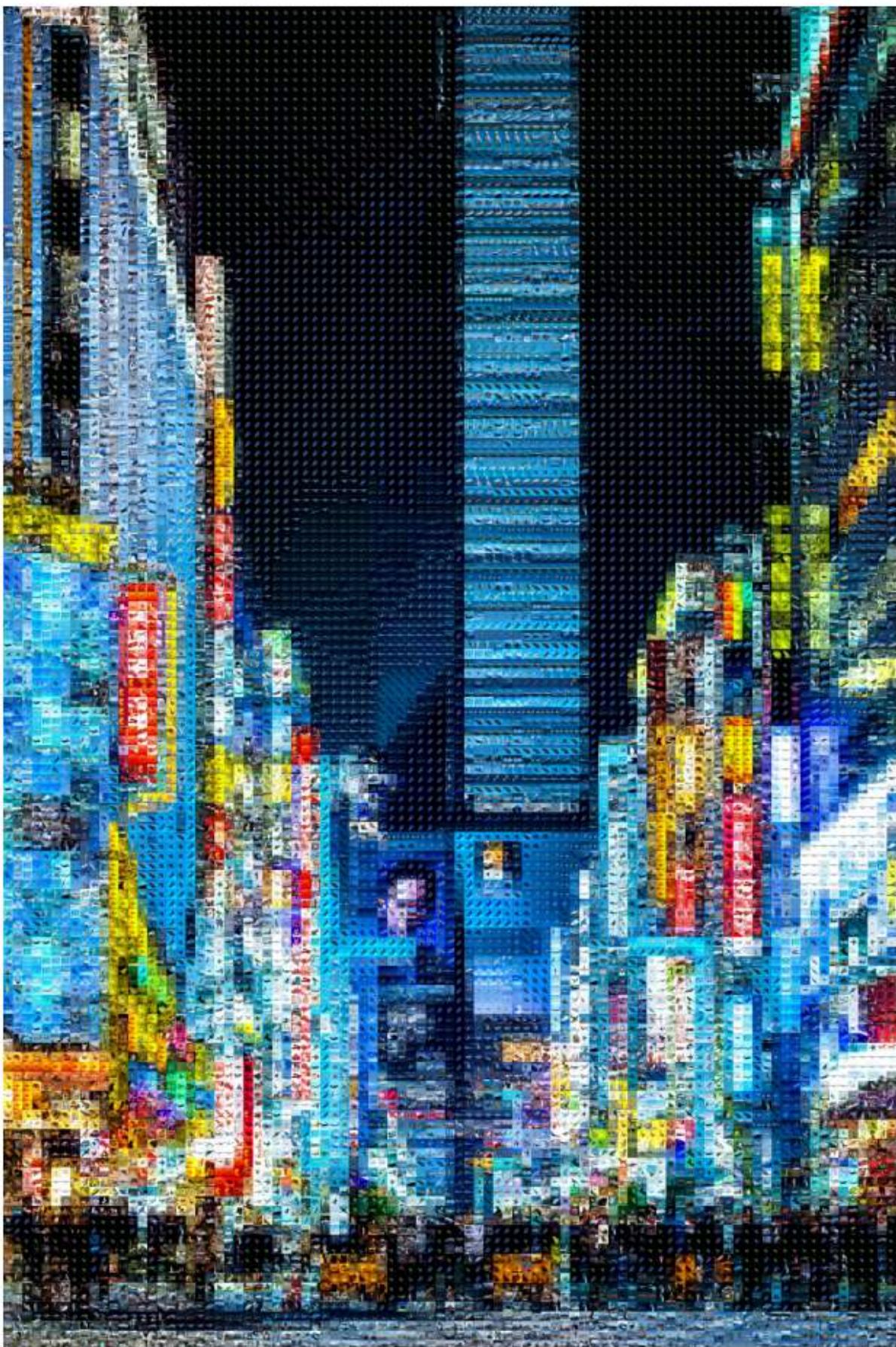
mosaic_combined = np.zeros_like(img)
idx = 0
for i in range(0, h, block_size):
    for j in range(0, w, block_size):
        thumb_idx = best_match_idx[idx]
        thumb = thumbnails[thumb_idx]
        thumb_resized = np.array(Image.fromarray(thumb).resize((block_size, block_size)))

        # Ajuste de color: coincidir media con bloque original
        block_mean = img[i:i+block_size, j:j+block_size].mean(axis=(0,1))
        thumb_mean = thumb_resized.mean(axis=(0,1))
        thumb_adjusted = thumb_resized + (block_mean - thumb_mean)
        thumb_adjusted = np.clip(thumb_adjusted, 0, 255)
```

```
mosaic_combined[i:i+block_size, j:j+block_size] = thumb_adjusted.astype(np.
idx += 1

plt.figure(figsize=(12, 12))
plt.imshow(mosaic_combined)
plt.axis("off")
plt.show()

# Guardar siquieres
Image.fromarray(mosaic_combined).save("content/LARGE IMAGE/photographic_mosaic_HSV_
```



Now metrics are established to compare the three results obtained:

```
In [45]: import matplotlib.pyplot as plt

# Crear la figura
fig, axes = plt.subplots(1, 4, figsize=(18, 6))

# Mostrar la imagen original
axes[0].imshow(img.astype(np.uint8))
axes[0].set_title("Original")
axes[0].axis("off")

# Mostrar mosaico por promedio RGB
axes[1].imshow(np.array(mosaic_avr).astype(np.uint8))
axes[1].set_title("Mosaic - Mean RGB")
axes[1].axis("off")

# Mostrar mosaico por histograma HSV
axes[2].imshow(np.array(mosaic_hsv).astype(np.uint8))
axes[2].set_title("Mosaic - HSV Histogram")
axes[2].axis("off")

# Mostrar mosaico combinado
axes[3].imshow(np.array(mosaic_combined).astype(np.uint8))
axes[3].set_title("Mosaic - Combined (RGB+HSV)")
axes[3].axis("off")

plt.tight_layout()
plt.show()
```



```
In [42]: from skimage.metrics import peak_signal_noise_ratio as psnr, structural_similarity
from skimage.color import rgb2lab, deltaE_ciede2000

# -----
# Función para evaluar un mosaico
# -----
def evaluate_mosaic(original, mosaic):
    # Asegurar formato float [0,1]
    orig = np.clip(original.astype(np.float32) / 255.0, 0, 1)
    mosa = np.clip(mosaic.astype(np.float32) / 255.0, 0, 1)

    # --- PSNR ---
    psnr_value = psnr(orig, mosa, data_range=1.0)
```

```

# --- SSIM ---
ssim_value = ssim(orig, mosa, channel_axis=2, data_range=1.0)

# --- ΔE (color perceptual) ---
orig_lab = rgb2lab(orig)
mosa_lab = rgb2lab(mosa)
delta_e = deltaE_ciede2000(orig_lab, mosa_lab)
mean_delta_e = np.mean(delta_e)

return psnr_value, ssim_value, mean_delta_e

# -----
# Evaluate all mosaics
# -----
metrics = {}
metrics["Mosaic - Mean RGB"] = evaluate_mosaic(img, np.array(mosaic_avr))
metrics["Mosaic - HSV Histogram"] = evaluate_mosaic(img, np.array(mosaic_hsv))
metrics["Mosaic - Combined (RGB+HSV+Adjust)"] = evaluate_mosaic(img, np.array(mosaic_rhav))

print("\n COMPARISON OF MOSAICS\n")
print(f"{'Method':>40s} {'PSNR':>10s} {'SSIM':>10s} {'ΔE (avg)':>12s}")
print("-*70")
for name, (psnr_val, ssim_val, delta_e) in metrics.items():
    print(f"{name:>40s} {psnr_val:10.3f} {ssim_val:10.3f} {delta_e:12.3f}")

# -----
# Choose the best according to criteria
# -----
best_psnr = max(metrics, key=lambda x: metrics[x][0])
best_ssim = max(metrics, key=lambda x: metrics[x][1])
best_color = min(metrics, key=lambda x: metrics[x][2])

print("\n🏆 Best PSNR:", best_psnr)
print("🏆 Best SSIM:", best_ssim)
print("🏆 Best Color Match (lowest ΔE):", best_color)

```

COMPARISON OF MOSAICS

Method	PSNR	SSIM	ΔE (avg)
<hr/>			
Mosaic - Mean RGB	11.531	0.117	19.010
Mosaic - HSV Histogram	10.022	0.145	22.124
Mosaic - Combined (RGB+HSV+Adjust)	11.622	0.131	18.411

- 🏆 Best PSNR: Mosaic - Combined (RGB+HSV+Adjust)
- 🏆 Best SSIM: Mosaic - HSV Histogram
- 🏆 Best Color Match (lowest ΔE): Mosaic - Combined (RGB+HSV+Adjust)