CSCA 5632 Final Project - Unsupervised and Supervised Learning on Animal Face Images (AFHQ Dataset)

By Moshiur Howlader

Github Link : https://github.com/Mosh333/csca5632-final-project

1. Introduction

In today's data-driven world, the ability to **uncover structure and meaning from unlabeled data** represents one of the most powerful and important areas in machine learning. While supervised learning depends on extensive labeled datasets, many real-world domains contain **vast quantities of raw, unannotated information**—such as images, text, medical scans, or sensor data—where manual labeling is costly or infeasible. Here, **unsupervised learning** plays a pivotal role: it enables algorithms to reveal hidden patterns, latent representations, and natural groupings within data without external supervision.

Unsupervised learning drives innovation across diverse domains:

- **Data exploration and pattern discovery:** Enables open-ended analysis of large, high-dimensional datasets to uncover hidden structures, correlations, and trends—reducing dimensionality and aiding human interpretation, such as exploring mass spectrometry data across large experimental datasets.

- **Computer vision:** Groups unlabeled images by similarity, compresses data via PCA, or learns visual embeddings through self-supervised methods like SimCLR.

- **Natural language processing:** Learns semantic relationships in text through Word2Vec or discovers latent topics using Latent Dirichlet Allocation (LDA).

- **Healthcare and biomedical research:** Facilitates the discovery of hidden disease patterns, comorbidity clusters, and patient subgroups from large-scale electronic health records—enabling better understanding of latent traits, risk domains, and disease progression, such as identifying novel comorbidity patterns in aging cohorts.

- **Autonomous systems and robotics:** Maps environments, groups sensor inputs, and learns spatial representations without labeled supervision.

- **Recommender and personalization systems:** Clusters users or content to generate recommendations when explicit ratings are unavailable.

Together, these examples highlight how unsupervised learning forms the foundation of **exploratory data analysis** and **representation learning**, allowing models to extract structure from raw data before labels exist.

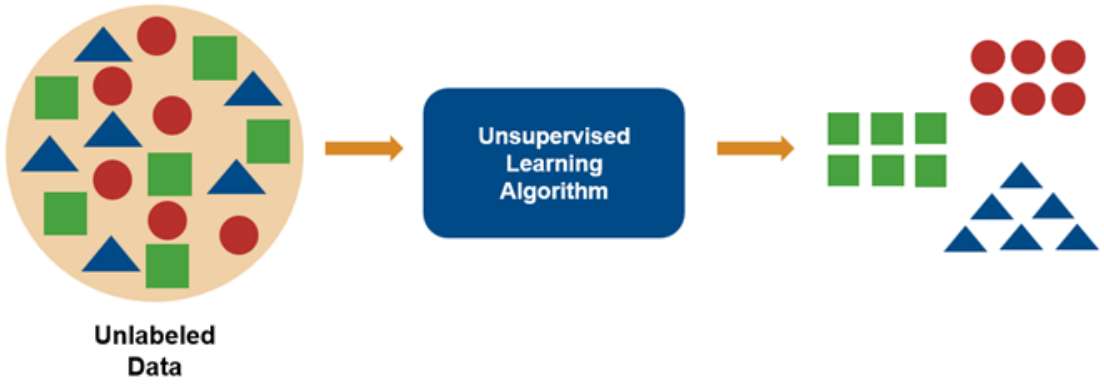


Figure: Conceptual illustration of unsupervised learning — an algorithm groups unlabeled data points (shapes) based on similarity, forming meaningful clusters.

1.1 Project Overview and Objectives

Here we discuss the selected data source and the unsupervised learning problem we aim to solve.

1.2 Gather Data, Determine the Method of Data Collection and Provenance

This project uses the **Animal Faces-HQ (AFHQ) dataset** — a publicly available image dataset originally curated by **Andrew Mvd** on Kaggle under a **CC BY-NC license**.
AFHQ contains **over 16,000 high-quality animal face images** across three balanced categories: **cats, dogs, and wildlife**.

According to the Kaggle description:

> "This dataset, also known as Animal Faces-HQ (AFHQ), consists of 16,130 high-quality images at 512×512 resolution.
> There are three domains of classes, each providing about 5000 images.
> By having multiple (three) domains and diverse images of various breeds per each domain, AFHQ sets a challenging image-to-image translation problem.
> The classes are: Cat, Dog, and Wildlife."

For this project, images are **resized to 128x128 pixels**, normalized to a [0, 1] range, and converted to **RGB tensors** (three-channel numerical arrays representing red, green, and blue intensities).
In addition to raw RGB features, three edge-based representations—**Canny**, **Sobel**, and **Laplacian**—are computed to evaluate how different hand-crafted feature types influence clustering performance.

These preprocessing steps prepare the data for feature extraction, dimensionality reduction, and clustering.
The dataset's high resolution, balance across categories, and visual diversity make it well-suited for evaluating **unsupervised image representation learning** and **clustering performance**.

Dataset Preview

Figure: Preview of the dataset used to perform this project.

1.3 Identify an Unsupervised Learning Problem

The goal of this project is to evaluate whether **unsupervised learning algorithms** can meaningfully **cluster animal face images** — cats, dogs, and wildlife — **based only on visual similarity**, without using any labels during training.
In other words, the objective is to determine whether these models can automatically group visually similar animals together.

The analysis includes **exploratory data analysis (EDA)**, **PCA-based dimensionality reduction**, and a comparison of multiple classical clustering algorithms: **K-Means**, **DBSCAN**, **Gaussian Mixture(GMM)** and **Agglomerative Clustering**.

Clustering performance is assessed using standard unsupervised metrics, including the **Adjusted Rand Index (ARI)**, and **Normalized Mutual Information (NMI)**, along with a cluster-to-label validation accuracy measure.

By comparing several unsupervised approaches—and contrasting their performance against a small supervised CNN baseline—this project highlights both the **capabilities and limitations** of classical clustering for image grouping tasks.
The results illustrate how different feature representations preserve or discard semantic information, and how clustering can reveal underlying **visual structure** in the dataset while motivating more powerful feature-learning methods.

---

2. Dataset Overview and Preprocessing

2.1 Fetching the Dataset

To begin, one must download the dataset (Github does not allow large data to be stored in a repo):

Git Bash / Linux / WSL:

```
curl -L -o "$(pwd)/data/animal-faces.zip" https://www.kaggle.com/api/v1/datas
ets/download/andrewmvd/animal-faces
```

After downloading, extract the dataset:

```
unzip "$(pwd)/data/animal-faces.zip" -d "$(pwd)/data/animal-faces"
```

To confirm successful extraction, verify that the dataset contains 16,130 images:

```
find "$(pwd)/data/animal-faces" -type f | wc -l
```

Expected output:

```
16130
```

Alternatively, one can simply download the image zip folder from https://www.kaggle.com/datasets/andrewmvd/animal-faces and store it under `~/data` and extract from there as `~/data/animal-faces` With the dataset successfully extracted and verified, the next step involves exploring its structure and visual characteristics through exploratory data analysis (EDA).

---

3. Exploratory Data Analysis (EDA)

3.1 Initial Inspection

This section inspects and visualizes the **Animal Faces-HQ (AFHQ)** dataset to understand its structure, quality, and key characteristics before model building.
The analysis focuses on data composition, visual patterns, feature correlations, preprocessing, and the main insights that will guide the subsequent unsupervised (and supervised) learning experiments.

Before applying clustering or dimensionality reduction, it is essential to perform an initial visual inspection of the dataset to gain intuition about its organization and diversity.

The dataset is organized into three main categories — **cats**, **dogs**, and **wildlife** — each containing roughly 5,000 high-quality 512×512 images.
Each category includes both training and validation subsets, stored under the following structure:

```
data/
├── animal-faces/                # Extracted dataset
│   ├── afhq/
│       ├── train/
│       │   ├── cat/             # ~5,153 images
│       │   ├── dog/             # ~4,739 images
│       │   └── wild/            # ~4,738 images
│       │
```

```
│       └── val/
│           ├── cat/            # 500 images
│           ├── dog/            # 500 images
│           └── wild/           # 500 images
│
└── animal-faces.zip            # Original downloaded dataset archive
```

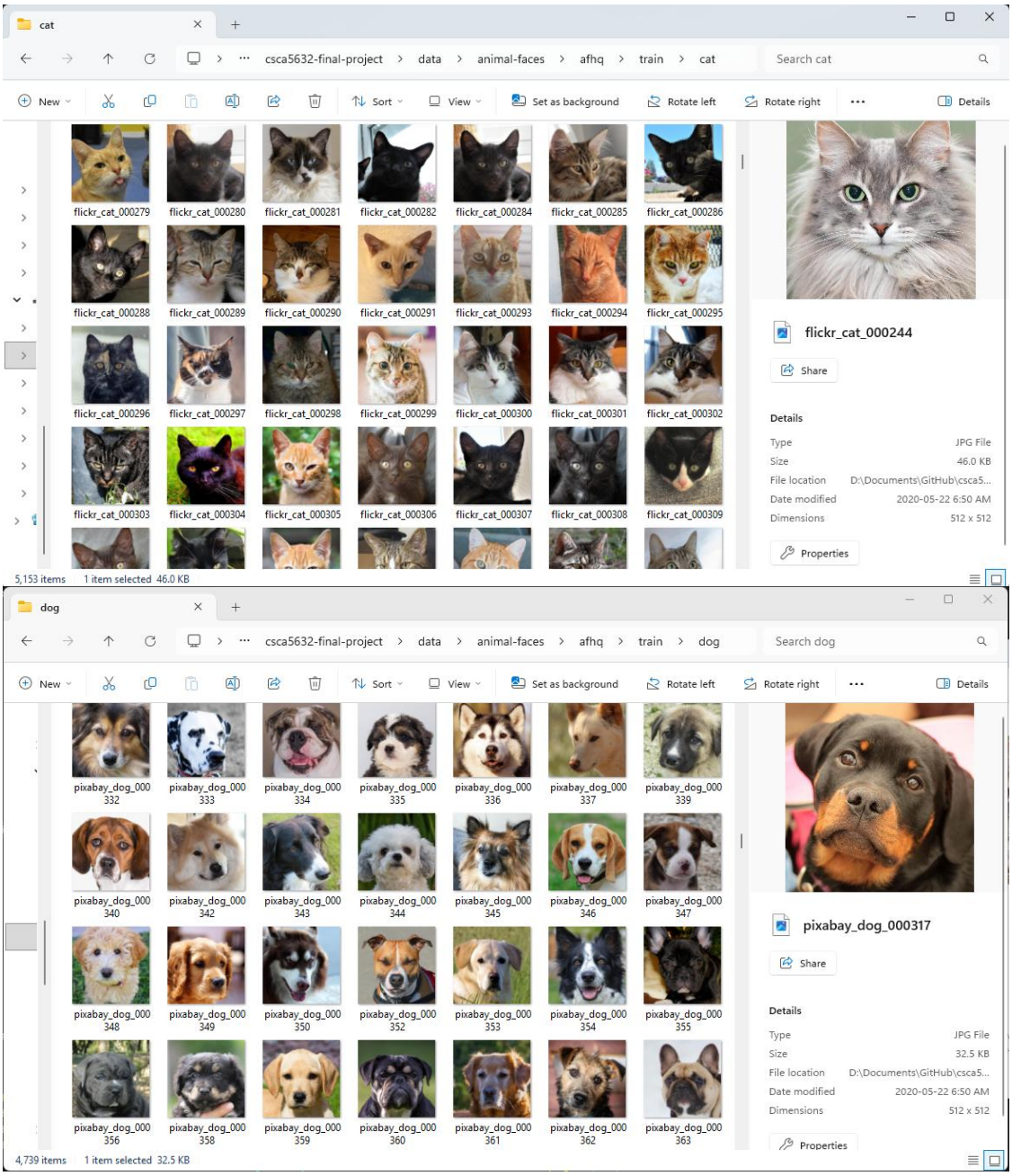
3.2 Visual Inspection

A few random samples from each class are shown below to demonstrate image quality and diversity.

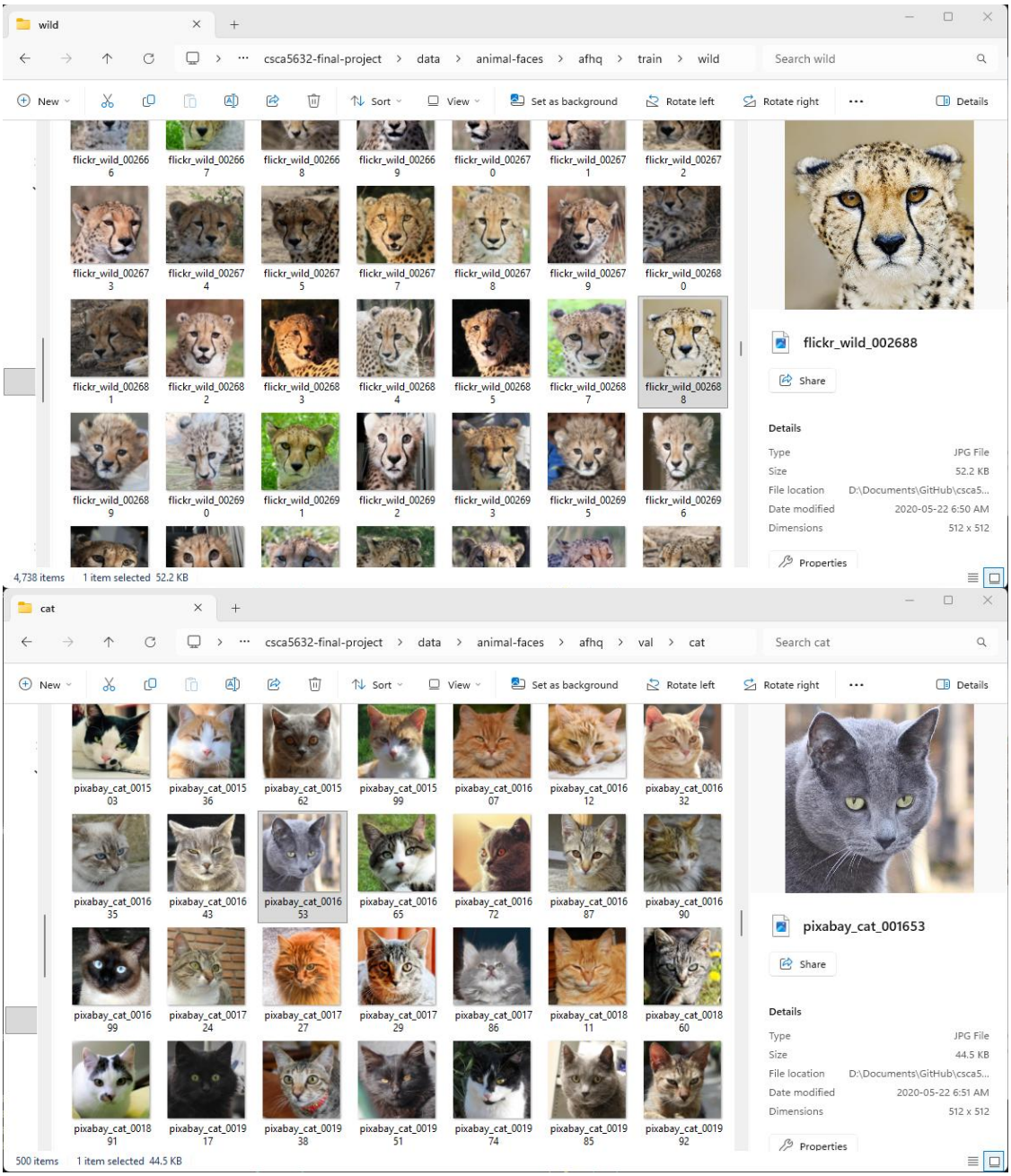
Observations:

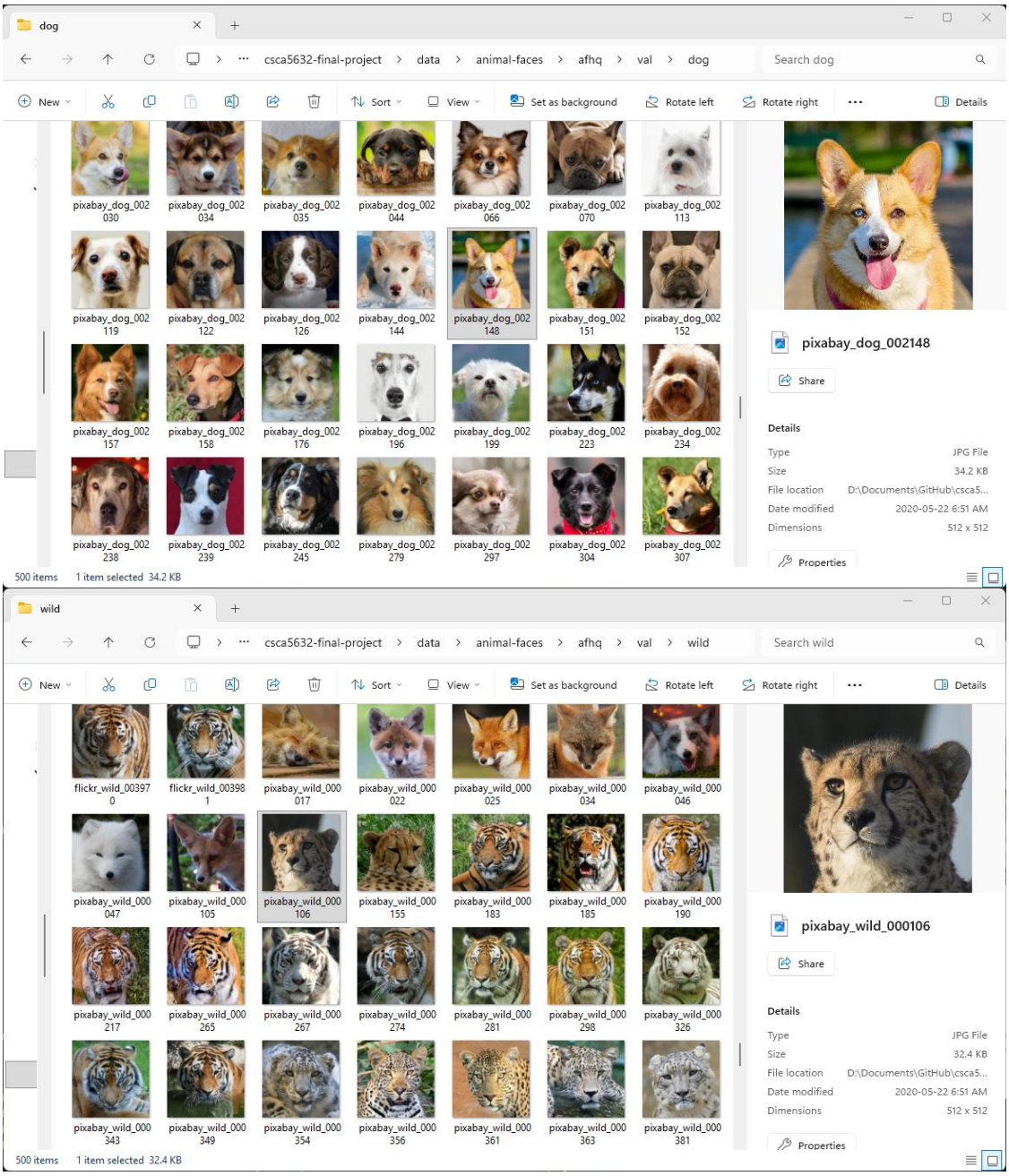
- The images are **balanced** across categories (≈5,000 per class).
- Each image is **centered and cropped** to focus on the animal's face.
- There is noticeable variation in lighting, background, and species within each class, which is beneficial for clustering and unsupervised generalization, as the algorithms are exposed to a richer set of visual features to learn from.

Moshiur Howlader



Moshiur Howlader





3.3 Dataset Composition and Descriptive Summary

Since this is the first project in this course involving image data, it is important to understand how the dataset is represented numerically before performing analysis.

Each image in the **AFHQ dataset** is a color image with a resolution of **512 × 512 pixels**, stored in the **RGB (Red, Green, Blue)** color model.
This means that every image is essentially a 3-dimensional array (or tensor) of shape **(512, 512, 3)**, where:

- The first two dimensions correspond to the image's **height** and **width** in pixels.
- The third dimension has **three channels** — one each for **red**, **green**, and **blue** color intensities.
- Each pixel location contains an RGB triplet, e.g. `[125, 64, 210]`, representing the color at that point (with intensity values typically ranging from 0–255).
- **0** corresponds to the absence of color intensity (black), while **255** represents maximum intensity (full brightness) for that channel.
- When all three channels are 0 (`[0, 0, 0]`), the pixel appears **black**, while when all are 255 (`[255, 255, 255]`), it appears **white**.

In total, each image contains:

- 512 × 512 = **262,144 pixels**
- Each pixel has 3 values -> **786,432 total intensity values per image**

These raw pixel intensities form the foundation for the statistical analysis, color distribution plots, and feature extraction steps that follow in this section.

For the EDA section, the NumPy and Panda library will be primarily used to explore insights into the data. For the model building sections, scikit-learn will be leveraged.

See the following for more details:

- **Understanding Digital Images for Image Processing and Computer Vision** (Medium)
- **RGB Color Model — Wikipedia**
- **Image Abstractions — MIT Computational Thinking**
- **Understanding Image Data Representation in Computer Systems** (DEV Community)
- **Digital Image Processing — Wikipedia**

3.3.1 Import Libraries, Load Dataset, and Perform Initial Data Summary

```
!pip install opencv-python
!pip install lz4
!pip install tensorflow
```

```
Requirement already satisfied: opencv-python in c:\users\howla\appdata\local\
programs\python\python312\lib\site-packages (4.12.0.88)
```

```
Requirement already satisfied: numpy<2.3.0,>=2 in c:\users\howla\appdata\loca
l\programs\python\python312\lib\site-packages (from opencv-python) (2.1.2)


[notice] A new release of pip is available: 24.0 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip

Requirement already satisfied: lz4 in c:\users\howla\appdata\local\programs\p
ython\python312\lib\site-packages (4.4.5)


[notice] A new release of pip is available: 24.0 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip

Requirement already satisfied: tensorflow in c:\users\howla\appdata\local\pro
grams\python\python312\lib\site-packages (2.20.0)
Requirement already satisfied: absl-py>=1.0.0 in c:\users\howla\appdata\local
\programs\python\python312\lib\site-packages (from tensorflow) (2.3.1)
Requirement already satisfied: astunparse>=1.6.0 in c:\users\howla\appdata\lo
cal\programs\python\python312\lib\site-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in c:\users\howla\appdata
\local\programs\python\python312\lib\site-packages (from tensorflow) (25.9.2
3)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in c:\user
s\howla\appdata\local\programs\python\python312\lib\site-packages (from tenso
rflow) (0.6.0)
Requirement already satisfied: google_pasta>=0.1.1 in c:\users\howla\appdata\
local\programs\python\python312\lib\site-packages (from tensorflow) (0.2.0)
Requirement already satisfied: libclang>=13.0.0 in c:\users\howla\appdata\loc
al\programs\python\python312\lib\site-packages (from tensorflow) (18.1.1)
Requirement already satisfied: opt_einsum>=2.3.2 in c:\users\howla\appdata\lo
cal\programs\python\python312\lib\site-packages (from tensorflow) (3.4.0)
Requirement already satisfied: packaging in c:\users\howla\appdata\local\prog
rams\python\python312\lib\site-packages (from tensorflow) (24.1)
Requirement already satisfied: protobuf>=5.28.0 in c:\users\howla\appdata\loc
al\programs\python\python312\lib\site-packages (from tensorflow) (6.33.1)
Requirement already satisfied: requests<3,>=2.21.0 in c:\users\howla\appdata\
local\programs\python\python312\lib\site-packages (from tensorflow) (2.32.3)
Requirement already satisfied: setuptools in c:\users\howla\appdata\local\pro
grams\python\python312\lib\site-packages (from tensorflow) (75.1.0)
Requirement already satisfied: six>=1.12.0 in c:\users\howla\appdata\local\pr
ograms\python\python312\lib\site-packages (from tensorflow) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in c:\users\howla\appdata\loc
al\programs\python\python312\lib\site-packages (from tensorflow) (3.2.0)
Requirement already satisfied: typing_extensions>=3.6.6 in c:\users\howla\app
data\local\programs\python\python312\lib\site-packages (from tensorflow) (4.1
5.0)
Requirement already satisfied: wrapt>=1.11.0 in c:\users\howla\appdata\local\
programs\python\python312\lib\site-packages (from tensorflow) (2.0.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in c:\users\howla\appdata\
```

```
local\programs\python\python312\lib\site-packages (from tensorflow) (1.76.0)
Requirement already satisfied: tensorboard~=2.20.0 in c:\users\howla\appdata\
local\programs\python\python312\lib\site-packages (from tensorflow) (2.20.0)
Requirement already satisfied: keras>=3.10.0 in c:\users\howla\appdata\local\
programs\python\python312\lib\site-packages (from tensorflow) (3.12.0)
Requirement already satisfied: numpy>=1.26.0 in c:\users\howla\appdata\local\
programs\python\python312\lib\site-packages (from tensorflow) (2.1.2)
Requirement already satisfied: h5py>=3.11.0 in c:\users\howla\appdata\local\p
rograms\python\python312\lib\site-packages (from tensorflow) (3.15.1)
Requirement already satisfied: ml_dtypes<1.0.0,>=0.5.1 in c:\users\howla\appd
ata\local\programs\python\python312\lib\site-packages (from tensorflow) (0.5.
4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in c:\users\howla\appdata\l
ocal\programs\python\python312\lib\site-packages (from astunparse>=1.6.0->ten
sorflow) (0.45.1)
Requirement already satisfied: rich in c:\users\howla\appdata\local\programs\
python\python312\lib\site-packages (from keras>=3.10.0->tensorflow) (14.2.0)
Requirement already satisfied: namex in c:\users\howla\appdata\local\programs
\python\python312\lib\site-packages (from keras>=3.10.0->tensorflow) (0.1.0)
Requirement already satisfied: optree in c:\users\howla\appdata\local\program
s\python\python312\lib\site-packages (from keras>=3.10.0->tensorflow) (0.18.
0)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\howla\app
data\local\programs\python\python312\lib\site-packages (from requests<3,>=2.2
1.0->tensorflow) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in c:\users\howla\appdata\local\p
rograms\python\python312\lib\site-packages (from requests<3,>=2.21.0->tensorf
low) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\howla\appdata\l
ocal\programs\python\python312\lib\site-packages (from requests<3,>=2.21.0->t
ensorflow) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\howla\appdata\l
ocal\programs\python\python312\lib\site-packages (from requests<3,>=2.21.0->t
ensorflow) (2024.8.30)
Requirement already satisfied: markdown>=2.6.8 in c:\users\howla\appdata\loca
l\programs\python\python312\lib\site-packages (from tensorboard~=2.20.0->tens
orflow) (3.10)
Requirement already satisfied: pillow in c:\users\howla\appdata\local\program
s\python\python312\lib\site-packages (from tensorboard~=2.20.0->tensorflow)
(10.4.0)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in c:\us
ers\howla\appdata\local\programs\python\python312\lib\site-packages (from ten
sorboard~=2.20.0->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in c:\users\howla\appdata\loca
l\programs\python\python312\lib\site-packages (from tensorboard~=2.20.0->tens
orflow) (3.1.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in c:\users\howla\appdata\lo
cal\programs\python\python312\lib\site-packages (from werkzeug>=1.0.1->tensor
board~=2.20.0->tensorflow) (2.1.5)
Requirement already satisfied: markdown-it-py>=2.2.0 in c:\users\howla\appdat
```

```
a\local\programs\python\python312\lib\site-packages (from rich->keras>=3.10.0
->tensorflow) (4.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in c:\users\howla\appd
ata\local\programs\python\python312\lib\site-packages (from rich->keras>=3.1
0.0->tensorflow) (2.18.0)
Requirement already satisfied: mdurl~=0.1 in c:\users\howla\appdata\local\pro
grams\python\python312\lib\site-packages (from markdown-it-py>=2.2.0->rich->k
eras>=3.10.0->tensorflow) (0.1.2)


[notice] A new release of pip is available: 24.0 -> 25.3
[notice] To update, run: python.exe -m pip install --upgrade pip
```

Load libraries and create helper functions:

```python
import os
import cv2
import numpy as np
import pandas as pd
from tqdm import tqdm
import matplotlib.pyplot as plt
import warnings
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
import psutil
import gc
from joblib import dump, load
import threading

# Get absolute path to the project root
BASE_DIR = os.path.abspath(os.path.join(os.getcwd(), ".."))  # go one level u
p from /notebooks
DATA_DIR = os.path.join(BASE_DIR, "data", "animal-faces", "afhq", "train")
VAL_DIR = os.path.join(BASE_DIR, "data", "animal-faces", "afhq", "val")
VAR_DATA_DIR = os.path.join(BASE_DIR, "data")  # For saving intermediate larg
e arrays

print("📂 Base directory:", BASE_DIR)
print("📁 Data directory:", DATA_DIR)
print("📁 Variable data directory:", VAR_DATA_DIR)

# === Helper Functions ===

def get_memory_usage():
    """Return total memory usage (GB) of the current process."""
    process = psutil.Process(os.getpid())
    return round(process.memory_info().rss / (1024 ** 3), 2)

def get_file_size(path):
```

```
    """Return file size in MB."""
    if os.path.exists(path):
        return round(os.path.getsize(path) / (1024 ** 2), 2)
    return 0

def memory_usage_gb(*arrays):
    """Compute total memory usage in GB for given numpy arrays."""
    total_bytes = sum(arr.nbytes for arr in arrays if isinstance(arr, np.ndar
ray))
    return round(total_bytes / (1024 ** 3), 2)

# def save_numpy_array(arr: np.ndarray, filename: str):
#     """Save numpy array with parallelized compression using joblib."""
#     filepath = os.path.join(VAR_DATA_DIR, filename.replace('.npz', '.pkl'))
#     dump(arr, filepath, compress=("lz4", 3))  # lz4 = super fast
#     print(f"⚙️ Saved {filename} with joblib-lz4 ({arr.nbytes / (1024**3):.
2f} GB) to {filepath}")

# === Fast async save using joblib + lz4 ===
def save_numpy_array_async(arr: np.ndarray, filename: str, compress_level: in
t = 3):
    """
    Save NumPy array asynchronously in background using joblib-lz4 compressio
n.
    Non-blocking, returns immediately while saving runs in a thread.
    """
    filename = os.path.splitext(filename)[0] + ".pkl"

    filepath = os.path.join(VAR_DATA_DIR, filename)
    filesize_gb = arr.nbytes / (1024 ** 3)

    def _save():
        try:
            dump(arr, filepath, compress=("lz4", compress_level))
            print(f"☑️ [Async Save Complete] {filename} ({filesize_gb:.2f} G
B) saved to {filepath}")
        except Exception as e:
            print(f"❌ [Async Save Error] {e}")

    thread = threading.Thread(target=_save, daemon=True)
    thread.start()

    print(f"⚙️ [Async Save Started] Saving {filename} ({filesize_gb:.2f} GB)
 in background...")
    return thread

def load_numpy_array(filename: str):
    """
```

```
    Load a previously saved NumPy array using joblib.
    Automatically looks inside VAR_DATA_DIR for the file.
    """
    filename = os.path.splitext(filename)[0] + ".pkl"

    filepath = os.path.join(VAR_DATA_DIR, filename)

    if not os.path.exists(filepath):
        print(f"❌ File not found: {filepath}")
        return None

    try:
        arr = load(filepath)
        print(f"☑ Loaded {filename} ({arr.nbytes / (1024**3):.2f} GB) from
{filepath}")
        return arr
    except Exception as e:
        print(f"❌ [Load Error] {e}")
        return None

def free_variable(var_name: str, globals_dict: dict):
    """Safely delete a variable and run garbage collection."""
    if var_name in globals_dict:
        del globals_dict[var_name]
        gc.collect()
        print(f"🧹 Deallocated variable: {var_name}")
    else:
        print(f"⚠ Variable '{var_name}' not found in memory.")

def wait_for_threads(*threads):
    """Wait for multiple async save threads to complete."""
    for t in threads:
        if t.is_alive():
            t.join()
    print("☑ All background saves completed.")
```

```
📁 Base directory: d:\Documents\GitHub\csca5632-final-project
📁 Data directory: d:\Documents\GitHub\csca5632-final-project\data\animal-fa
ces\afhq\train
📁 Variable data directory: d:\Documents\GitHub\csca5632-final-project\data
```

Load the actual images as data:

```
# === Memory before loading ===
print(f"💡 Memory usage before loading dataset: {get_memory_usage()} GB\n")

# Get all class names (cat, dog, wild)
```

```
classes = os.listdir(DATA_DIR)
print("Classes:", classes)

# Lists for data and labels
data = []
labels_train = []
records = []  # For summary DataFrame
missing = 0

# Modern OS automatically cache recently read files in RAM for faster speed u
p on re-runs
# Resize image to be smaller on systems with less than 32GB RAM (mine is 64GB
 DDR5)
for cls in classes:
    folder = os.path.join(DATA_DIR, cls)
    for fname in tqdm(os.listdir(folder), desc=f"Loading {cls}"):
        fpath = os.path.join(folder, fname)
        img = cv2.imread(fpath)
        if img is None:
            print(f"⚠ Skipping corrupted file: {fname}")
            missing += 1
            continue
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        # img = cv2.resize(img, (128, 128))   # optional resize for speed, on
ly uncomment if system is less than 32GB RAM

        # Append image and label for modeling
        data.append(img)
        labels_train.append(cls)

        # Record summary stats for DataFrame
        records.append({
            'File': fname,
            'Class': cls,
            'Height': img.shape[0],
            'Width': img.shape[1],
            'Mean_R': np.mean(img[:, :, 0]),
            'Mean_G': np.mean(img[:, :, 1]),
            'Mean_B': np.mean(img[:, :, 2])
        })

# Convert to numpy arrays for ML
data = np.array(data)
labels_train = np.array(labels_train)

# Create summary DataFrame
df_summary = pd.DataFrame(records)

# === Memory after loading ===
```

```python
print(f"\n💡 Memory usage after loading dataset: {get_memory_usage()} GB")

print("☑️ Data shape:", data.shape)
print("☑️ Labels shape:", labels_train.shape)
print("☑️ Summary DataFrame shape:", df_summary.shape)
print(df_summary['Class'].value_counts())
print("Min pixel value:", data.min())
print("Max pixel value:", data.max())
print(f"Missing or unreadable files: {missing}")

# === Optional: Save raw data for later use ===
save_thread_rgb = save_numpy_array_async(data, "AFHQ_RGB_dataset")

# === Safe to free memory after starting async saves — the background thread keeps its own reference ===
# free_variable("data", globals())

# === Display summary preview ===
display(df_summary.head())
display(df_summary.describe())
```

```
💡 Memory usage before loading dataset: 0.19 GB

Classes: ['cat', 'dog', 'wild']

Loading cat: 100%|██████████| 5153/5153 [00:07<00:00, 657.52it/s]
Loading dog: 100%|██████████| 4739/4739 [00:07<00:00, 662.02it/s]
Loading wild: 100%|██████████| 4738/4738 [00:07<00:00, 627.35it/s]


💡 Memory usage after loading dataset: 10.92 GB
✅ Data shape: (14630, 512, 512, 3)
✅ Labels shape: (14630,)
✅ Summary DataFrame shape: (14630, 7)
Class
cat     5153
dog     4739
wild    4738
Name: count, dtype: int64
Min pixel value: 0
Max pixel value: 255
Missing or unreadable files: 0
⚙️ [Async Save Started] Saving AFHQ_RGB_dataset.pkl (10.72 GB) in backgroun
d...
```

	File	Class	Height	Width	Mean_R	Mean_G	\
0	flickr_cat_000002.jpg	cat	512	512	39.049721	37.773727	
1	flickr_cat_000003.jpg	cat	512	512	116.653389	105.813721	
2	flickr_cat_000004.jpg	cat	512	512	110.633583	104.424599	

```
3  flickr_cat_000005.jpg   cat     512    512   93.215023   96.443832
4  flickr_cat_000006.jpg   cat     512    512   93.932934  100.388748

       Mean_B
0   32.619888
1   91.708538
2   97.659111
3  103.763954
4  108.864693

         Height    Width        Mean_R        Mean_G        Mean_B
count  14630.0  14630.0  14630.000000  14630.000000  14630.000000
mean     512.0    512.0    128.056264    117.270526    101.834385
std        0.0      0.0     28.003640     25.726367     28.416515
min      512.0    512.0     12.029060     19.568592     16.379856
25%      512.0    512.0    109.640875    100.432546     82.481689
50%      512.0    512.0    127.981014    116.936129    100.384445
75%      512.0    512.0    146.396766    133.766340    119.739083
max      512.0    512.0    241.809887    231.551273    231.936943

✅ [Async Save Complete] AFHQ_RGB_dataset.pkl (10.72 GB) saved to d:\Documen
ts\GitHub\csca5632-final-project\data\AFHQ_RGB_dataset.pkl
```

💡 **Explanation of code above:**

The AFHQ dataset contains a total of 14,630 color images divided across three balanced categories:

- Cat: 5,153 images
- Dog: 4,739 images
- Wild: 4,738 images

Each image has a fixed resolution of 512 × 512 pixels, confirming uniform dimensions across the dataset. This consistency simplifies preprocessing and model training by ensuring identical tensor shapes.

The pixel intensity features (`Mean_R`, `Mean_G`, and `Mean_B`) show reasonable variation across samples, with values ranging approximately from 12 to 241 for the red channel, 16 to 231 for green, and 16 to 231 for blue.

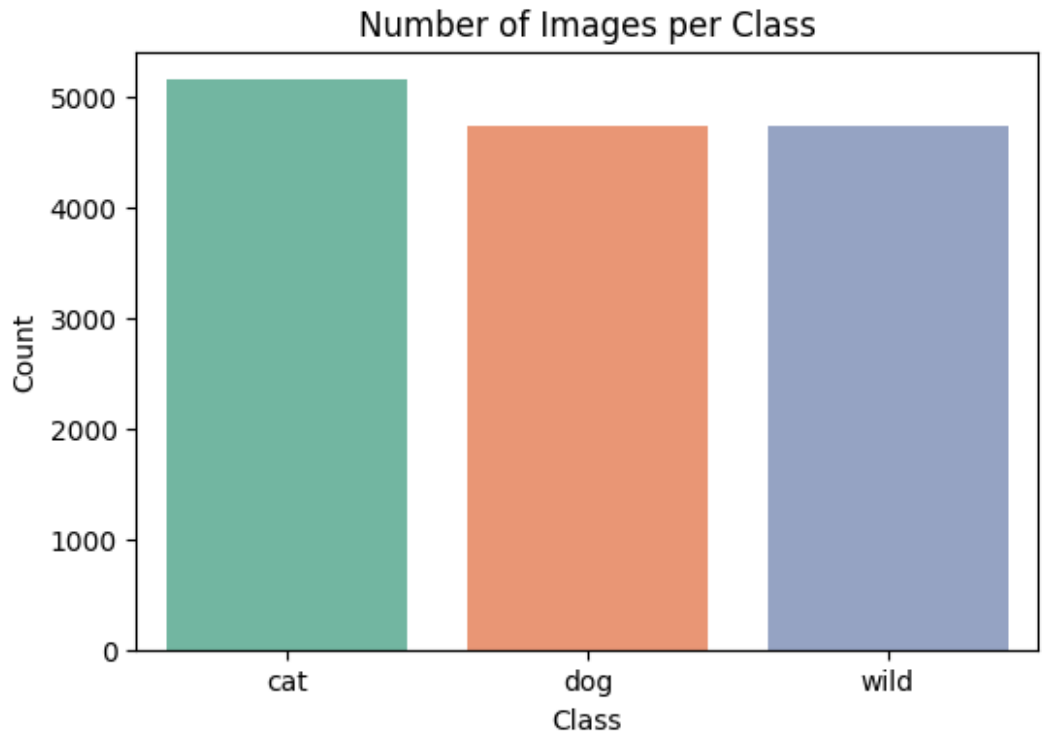
The mean RGB intensities are around `R: 128.1`, `G: 117.7`, and `B: 101.8`, suggesting that the images overall have slightly warmer color tones (stronger red component).

The standard deviations (`std ≈ 28`) indicate moderate variability in color brightness across images, which is expected in natural animal photographs with varying brightness.

No missing or corrupted files were detected, and all pixel values fall within the valid 0–255 range, confirming that the dataset is clean and ready for analysis

3.3.2 Class Distribution Overview

```
import pandas as pd
import seaborn as sns

warnings.filterwarnings("ignore", category=FutureWarning)
class_counts = pd.Series(labels_train).value_counts()
plt.figure(figsize=(6,4))
sns.barplot(x=class_counts.index, y=class_counts.values, palette="Set2")
plt.title("Number of Images per Class")
plt.xlabel("Class"); plt.ylabel("Count")
plt.show()

print(class_counts)
```



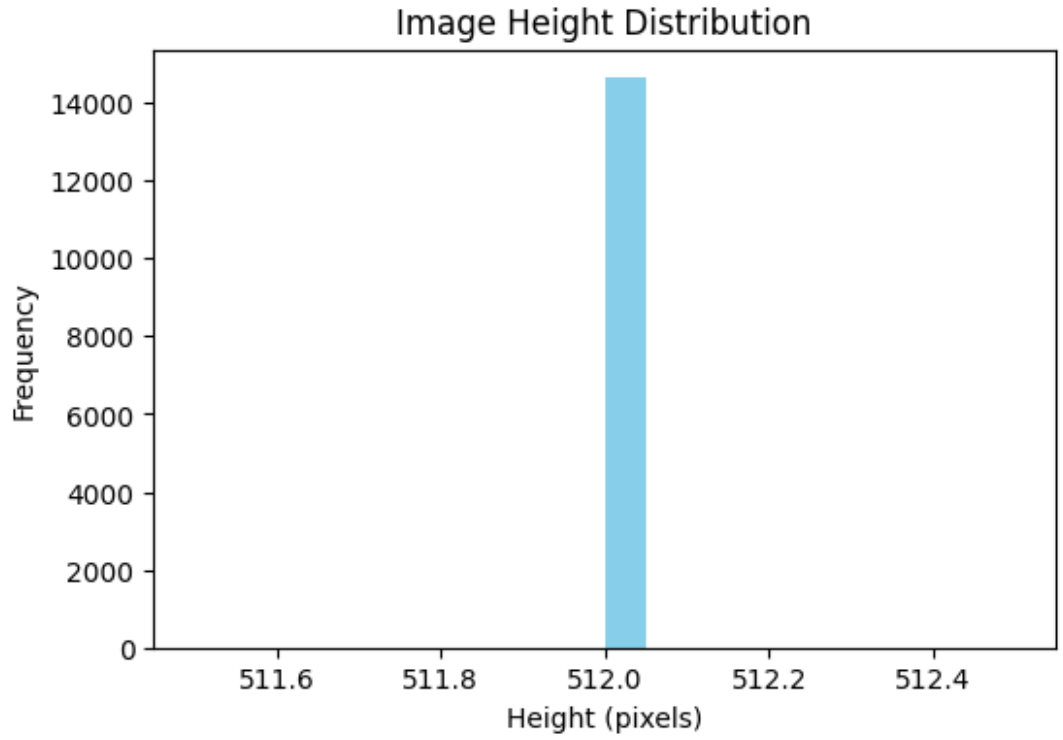

```
cat     5153
dog     4739
wild    4738
Name: count, dtype: int64
```

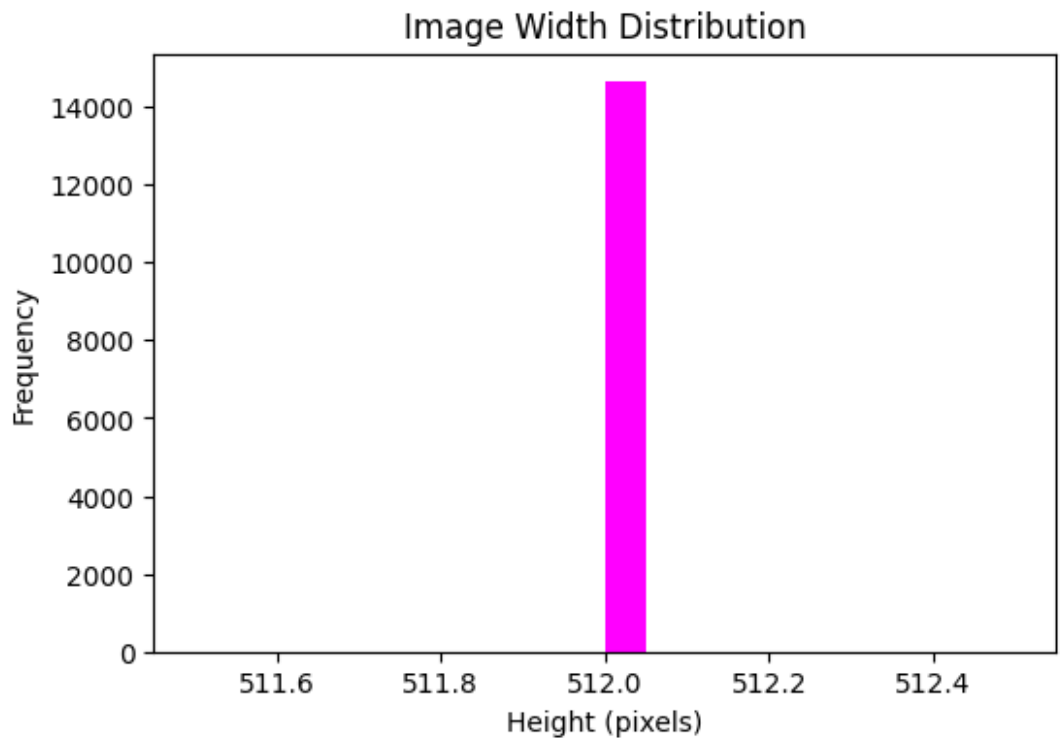
💡 **Explanation of code above:**

This code calculates the number of images in each category (`cat`, `dog`, and `wild`) using a pandas `value_counts()` function and visualizes the class distribution with a Seaborn bar plot. The output shows that the dataset is well-balanced, with approximately 5,000 images per class. This balance ensures that no class dominates the analysis and helps maintain fairness during unsupervised learning or clustering.

3.3.3 Image dimensions

```
heights, widths = zip(*[img.shape[:2] for img in data])
plt.figure(figsize=(6,4))
plt.hist(heights, bins=20, color='skyblue')
plt.title("Image Height Distribution")
plt.xlabel("Height (pixels)")
plt.ylabel("Frequency")
plt.show()

plt.figure(figsize=(6,4))
plt.hist(widths, bins=20, color='magenta')
plt.title("Image Width Distribution")
plt.xlabel("Height (pixels)")
plt.ylabel("Frequency")
plt.show()
```



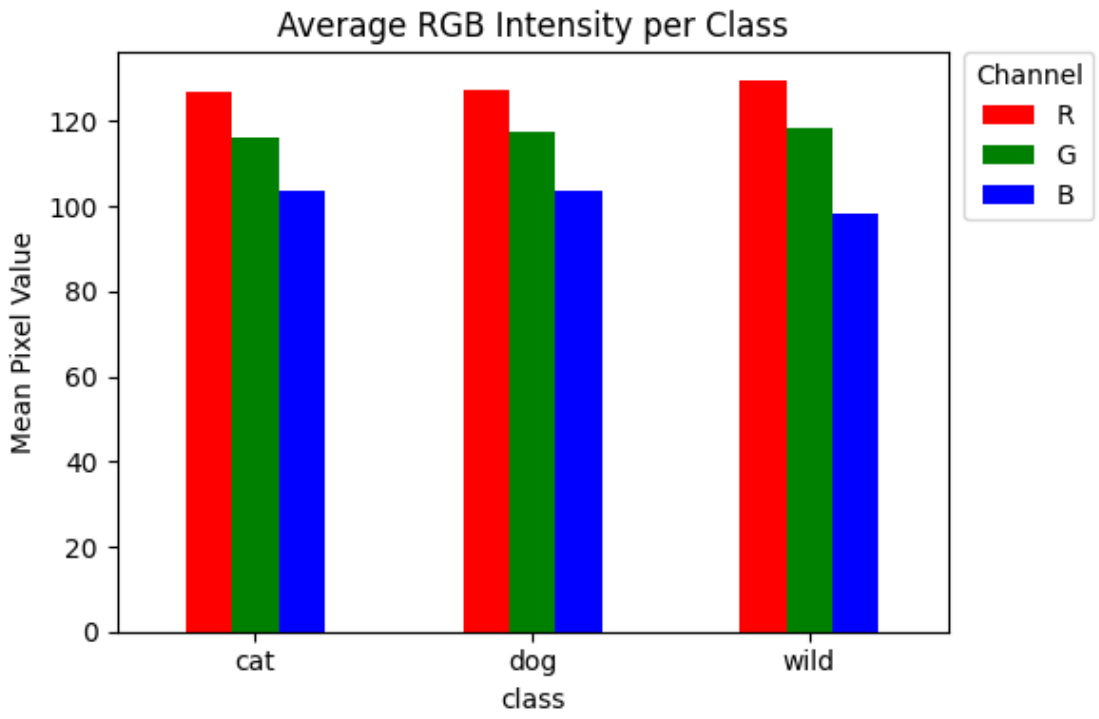


💡 **Explanation of code above:**
Both histograms confirm that all images have a uniform resolution of 512 × 512 pixels — an important property for consistent feature extraction and model input.

3.3.4 RGB channel intensities per class

```
means = {'class': [], 'R': [], 'G': [], 'B': []}

for cls in np.unique(labels_train):
    imgs = np.array([data[i] for i in range(len(data)) if labels_train[i] ==
cls])
    mean_rgb = imgs.mean(axis=(0, 1, 2))
    means['class'].append(cls)
    means['R'].append(mean_rgb[0])
    means['G'].append(mean_rgb[1])
    means['B'].append(mean_rgb[2])

df_means = pd.DataFrame(means)

# Plot with cleaner layout
ax = df_means.set_index('class').plot(
    kind='bar',
    figsize=(6, 4),
    color=['r', 'g', 'b'],
    legend=True
)
```

```
plt.title("Average RGB Intensity per Class")
plt.ylabel("Mean Pixel Value")
plt.xticks(rotation=0)  # keeps labels horizontal
plt.legend(title="Channel", bbox_to_anchor=(1.02, 1), loc='upper left', borde
raxespad=0)
plt.tight_layout()  # adjusts spacing to fit legend cleanly
plt.show()
```

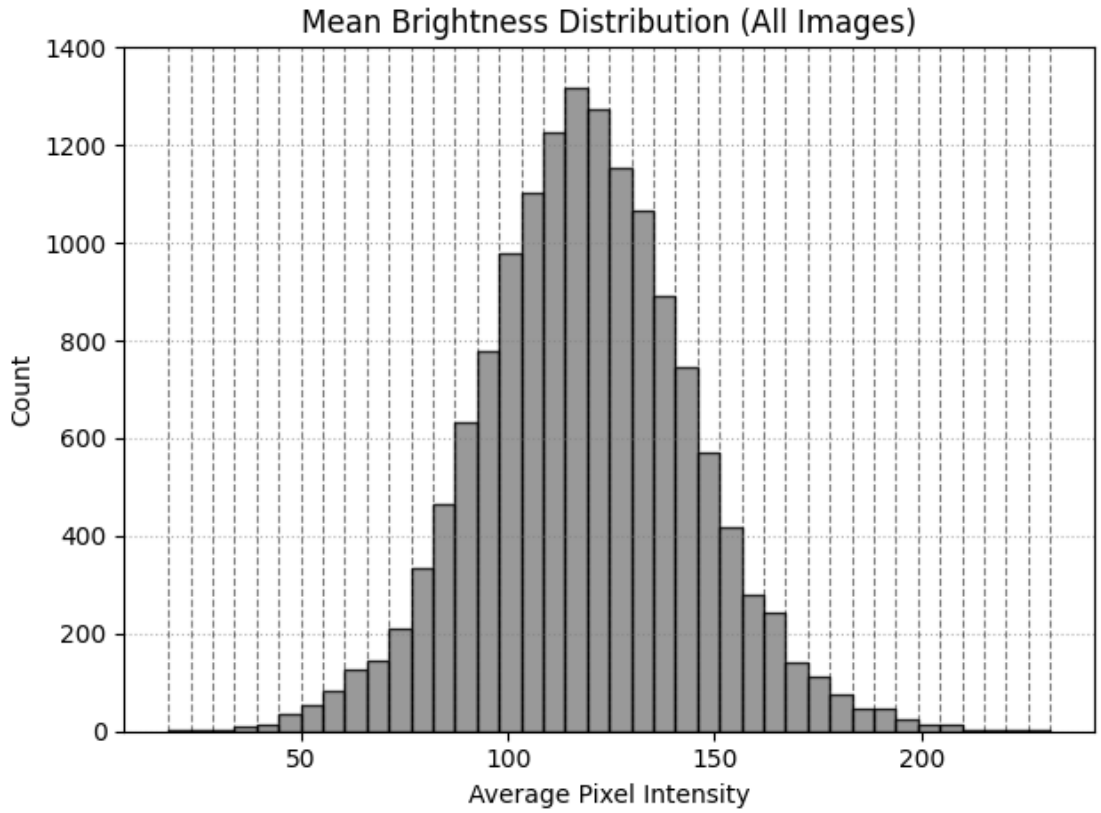


💡 **Explanation of code above:**
This bar chart shows the average RGB channel intensities for each class, indicating that the dataset's images are generally warmer in tone, with slightly higher red intensity values compared to green and blue.

3.3.5 Brightness / grayscale distribution

```
gray_means = []

for img in data:
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    gray_means.append(gray.mean())

# Plot histogram and capture bin data
counts, bins, patches = plt.hist(gray_means, bins=40, color='gray', edgecolor
='black', alpha=0.8)

# Draw vertical bin lines
for b in bins:
    plt.axvline(x=b, color='black', linestyle='--', linewidth=0.8, alpha=0.5)
```

```
# Add horizontal gridlines
plt.grid(axis='y', linestyle=':', color='gray', alpha=0.6)

# Force specific y-ticks (add 1400)
y_ticks = list(plt.yticks()[0])          # get current ticks
if 1400 not in y_ticks:
    y_ticks.append(1400)                 # add 1400 manually
y_ticks = sorted(set(y_ticks))           # remove duplicates and sort
plt.yticks(y_ticks)

plt.title("Mean Brightness Distribution (All Images)")
plt.xlabel("Average Pixel Intensity")
plt.ylabel("Count")
plt.tight_layout()
plt.show()
```



💡 **Explanation of code above:**
This histogram illustrates the distribution of **mean brightness** values across all images in the dataset.
In digital imaging, **brightness** refers to the *average luminance or pixel intensity* of an image — a measure of how light or dark an image appears when converted to grayscale, where pixel values range from **0 (black)** to **255 (white)**.

In grayscale conversion, each pixel's intensity is computed as a weighted sum of its RGB components:

$$Y = 0.299R + 0.587G + 0.114B$$

where the constants correspond to the standard luminance coefficients defined in the **OpenCV** implementation.

Most images in the dataset have mean brightness values between **90 and 150**, forming a near-normal, bell-shaped curve centered around **~120**. This indicates that the dataset contains **well-exposed images** with balanced illumination — neither underexposed nor overexposed. Such uniform brightness ensures that lighting variations do not bias feature learning, resulting in more consistent and robust model performance during unsupervised analysis.

Sources:

- Digital Image Processing — Brightness and Contrast (TutorialsPoint)
- Grayscale — Wikipedia
- OpenCV Documentation — Color Conversions
- Stackover Flow Discussion on Luminance

Summary of Section 3.3 — Dataset Composition and Descriptive Summary

This section performed an exploratory analysis of the AFHQ dataset to understand its composition, structure, and fundamental visual properties before applying unsupervised learning methods.

- **Dataset Overview:** The dataset contains a total of 14,630 color images evenly distributed across three balanced categories — Cat (5,153), Dog (4,739), and Wild (4,738) — ensuring no class imbalance.
- **Image Structure:** Each image is a 512 × 512 RGB tensor, representing 786,432 total pixel values per image. The resolution is perfectly consistent across all files, simplifying preprocessing and guaranteeing compatibility for downstream feature extraction and clustering.
- **Color and Intensity Distribution:** The average RGB channel means were approximately R = 128.1, G = 117.7, and B = 101.8, indicating that images tend to have slightly warmer tones (mild red dominance). Pixel intensity ranges and standard deviations (≈ 28) revealed moderate variability in color brightness, typical of naturally lit animal photographs.
- **Brightness and Exposure:** The mean brightness histogram (computed from grayscale luminance `Y = 0.299R + 0.587G + 0.114B`) showed a bell-shaped

distribution centered around ~120, confirming that most images are well-exposed with balanced illumination. This uniform exposure minimizes the influence of lighting differences on model training.

- **Data Integrity:** No missing or corrupted files were detected, and all pixel values lie within the expected 0–255 range, confirming that the dataset is clean and ready for analysis.

Overall, the EDA confirms that the AFHQ dataset is well-balanced, high-quality, and visually consistent, providing a strong foundation for unsupervised learning tasks such as feature extraction, clustering, or representation learning.

3.4 Feature Correlations and Visual Patterns

In this section, we explore how the RGB color channels relate to each other and what visual patterns exist across the three animal classes. The goal is to see if certain color channels are strongly correlated, if there are any noticeable outliers, and whether different classes show unique color intensity trends. These insights help us understand how consistent the dataset's color information is and whether any preprocessing or normalization might be needed before applying unsupervised learning methods.

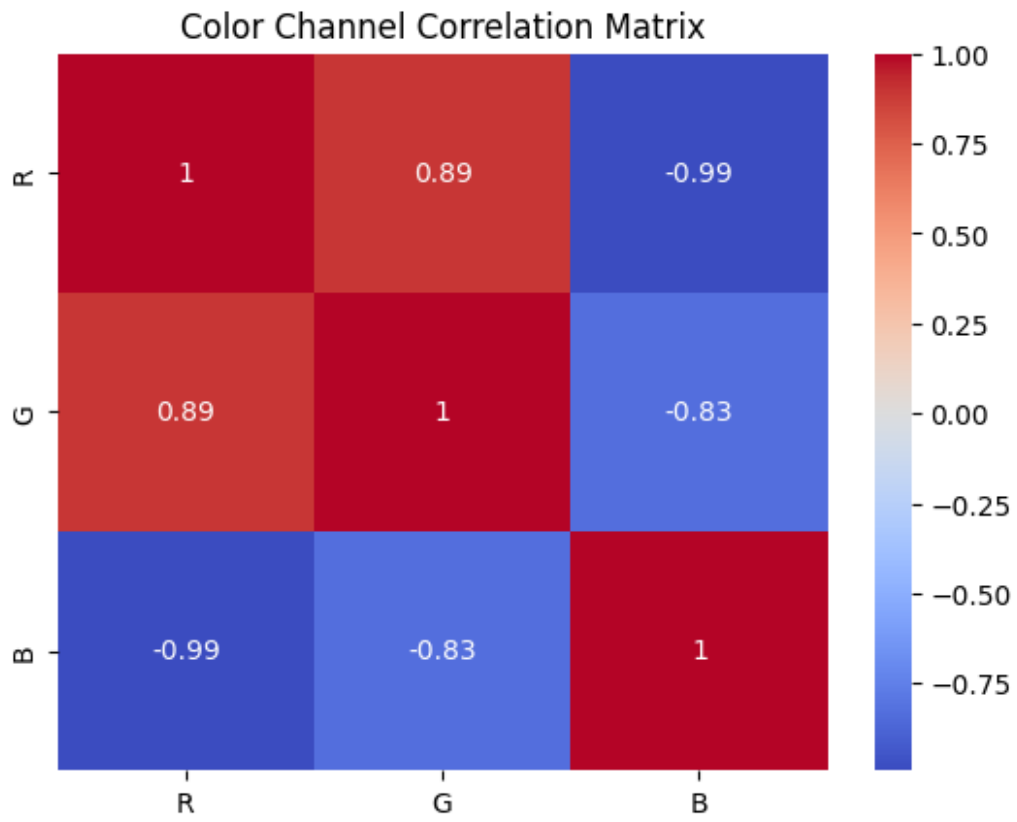
```
corr = df_means[['R','G','B']].corr()
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title("Color Channel Correlation Matrix")
plt.show()

df_summary[['Mean_R','Mean_G','Mean_B']].plot.box(figsize=(6,4), color='black
')
plt.title("Outlier Check for Average Color Intensity per Channel")
plt.ylabel("Mean Pixel Value")
plt.show()


# Custom pastel palette (purple, pink, gold)
custom_palette = ["#9b5de5", "#f15bb5", "#fee440"]

sns.scatterplot(data=df_summary, x='Mean_R', y='Mean_G', hue='Class', alpha=
0.5)
plt.title("Mean Red vs Green Intensities by Class")
plt.show()


fig, axes = plt.subplots(1, 3, figsize=(15,4), sharex=True, sharey=True)
for ax, (cls, color) in zip(axes, zip(classes, custom_palette)):
    subset = df_summary[df_summary['Class'] == cls]
    sns.scatterplot(
```

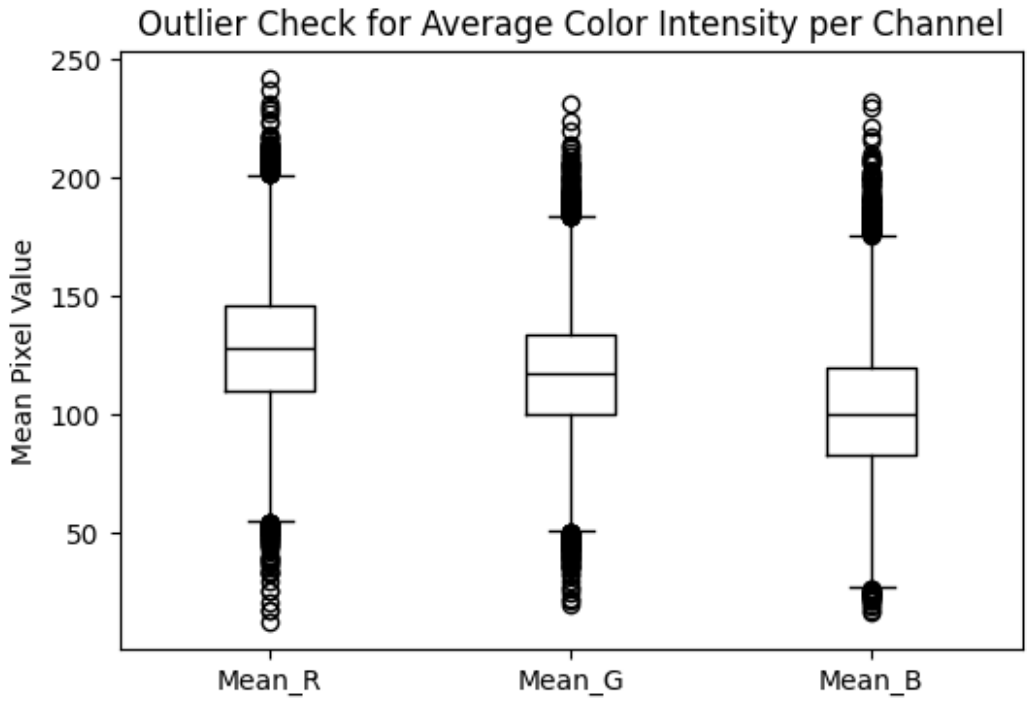
```
        data=subset, x='Mean_R', y='Mean_G',
        alpha=0.5, ax=ax, color=color, edgecolor='none'
    )
    ax.set_title(f"{cls.capitalize()} — R vs G", fontsize=12, color=color)
    ax.set_xlabel("Mean_R")
    ax.set_ylabel("Mean_G")
    ax.grid(True, linestyle=":", alpha=0.4)
plt.suptitle("Mean Red vs Green Intensities by Class", fontsize=14, weight='b
old')
plt.tight_layout()
plt.show()

# Mean Red vs Blue Intensities by Class
sns.scatterplot(data=df_summary, x='Mean_R', y='Mean_B', hue='Class', alpha=
0.5)
plt.title("Mean Red vs Blue Intensities by Class")
plt.xlabel("Mean_R")
plt.ylabel("Mean_B")
plt.show()

# Per-class subplots (R vs B)
fig, axes = plt.subplots(1, 3, figsize=(15,4), sharex=True, sharey=True)
for ax, (cls, color) in zip(axes, zip(classes, custom_palette)):
    subset = df_summary[df_summary['Class'] == cls]
    sns.scatterplot(
        data=subset, x='Mean_R', y='Mean_B',
        alpha=0.5, ax=ax, color=color, edgecolor='none'
    )
    ax.set_title(f"{cls.capitalize()} — R vs B", fontsize=12, color=color)
    ax.set_xlabel("Mean_R")
    ax.set_ylabel("Mean_B")
    ax.grid(True, linestyle=":", alpha=0.4)
plt.suptitle("Mean Red vs Blue Intensities by Class", fontsize=14, weight='bo
ld')
plt.tight_layout()
plt.show()

# Mean Green vs Blue Intensities by Class
sns.scatterplot(data=df_summary, x='Mean_G', y='Mean_B', hue='Class', alpha=
0.5)
plt.title("Mean Green vs Blue Intensities by Class")
plt.xlabel("Mean_G")
plt.ylabel("Mean_B")
plt.show()

# Per-class subplots (G vs B)
fig, axes = plt.subplots(1, 3, figsize=(15,4), sharex=True, sharey=True)
for ax, (cls, color) in zip(axes, zip(classes, custom_palette)):
    subset = df_summary[df_summary['Class'] == cls]
```

```
    sns.scatterplot(
        data=subset, x='Mean_G', y='Mean_B',
        alpha=0.5, ax=ax, color=color, edgecolor='none'
    )
    ax.set_title(f"{cls.capitalize()} — G vs B", fontsize=12, color=color)
    ax.set_xlabel("Mean_G")
    ax.set_ylabel("Mean_B")
    ax.grid(True, linestyle=":", alpha=0.4)
plt.suptitle("Mean Green vs Blue Intensities by Class", fontsize=14, weight='
bold')
plt.tight_layout()
plt.show()


# Compute grayscale brightness using standard luminance coefficients
df_summary['Brightness'] = 0.299*df_summary['Mean_R'] + 0.587*df_summary['Mea
n_G'] + 0.114*df_summary['Mean_B']

# Correlation matrix including brightness
# corr_extended = df_summary[['Mean_R', 'Mean_G', 'Mean_B', 'Brightness']].co
rr()


# Plot individual channel distributions per class
fig, axes = plt.subplots(1, 3, figsize=(15,4), sharey=True)

channels = ['Mean_R', 'Mean_G', 'Mean_B']
titles = ['Distribution of Mean Red Intensity',
          'Distribution of Mean Green Intensity',
          'Distribution of Mean Blue Intensity']

for ax, ch, title in zip(axes, channels, titles):
    sns.kdeplot(
        data=df_summary,
        x=ch,
        hue='Class',
        fill=True,
        common_norm=False,
        palette=custom_palette,
        alpha=0.5,
        ax=ax
    )
    ax.set_title(title, fontsize=12)
    ax.set_xlabel("Mean Pixel Value")
    ax.set_ylabel("Density")
    ax.grid(True, linestyle=":", alpha=0.4)

plt.suptitle("Color Channel Distributions Across Classes", fontsize=14, weigh
t='bold')
```

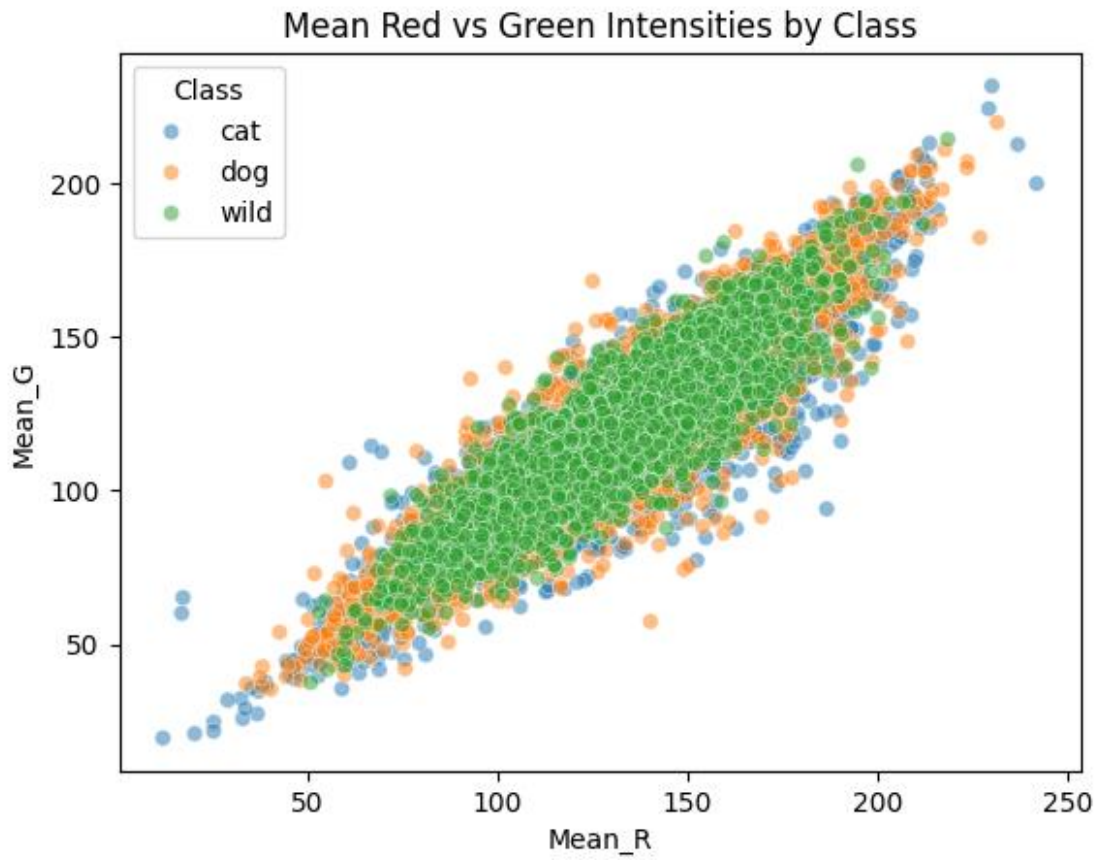
```
plt.tight_layout()
plt.show()


# Pairplot for RGB means
sns.pairplot(df_summary[['Mean_R', 'Mean_G', 'Mean_B', 'Class']], hue='Class
', diag_kind='kde', palette=custom_palette)
plt.suptitle("Pairwise Relationships Among RGB Channels by Class", y=1.02)
plt.show()



fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111, projection='3d')

colors = {'cat': '#9b5de5', 'dog': '#f15bb5', 'wild': '#fee440'}

for cls in classes:
    subset = df_summary[df_summary['Class'] == cls]
    ax.scatter(subset['Mean_R'], subset['Mean_G'], subset['Mean_B'],
               color=colors[cls], label=cls, alpha=0.5, s=15)

ax.set_xlabel('Mean_R')
ax.set_ylabel('Mean_G')
ax.set_zlabel('Mean_B')
ax.set_title("3D RGB Color Distribution by Class")
ax.legend()
plt.tight_layout()
plt.show()
```

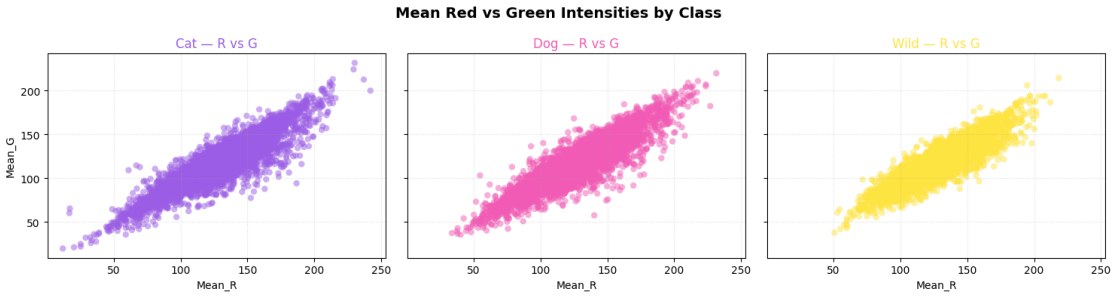
Moshiur Howlader



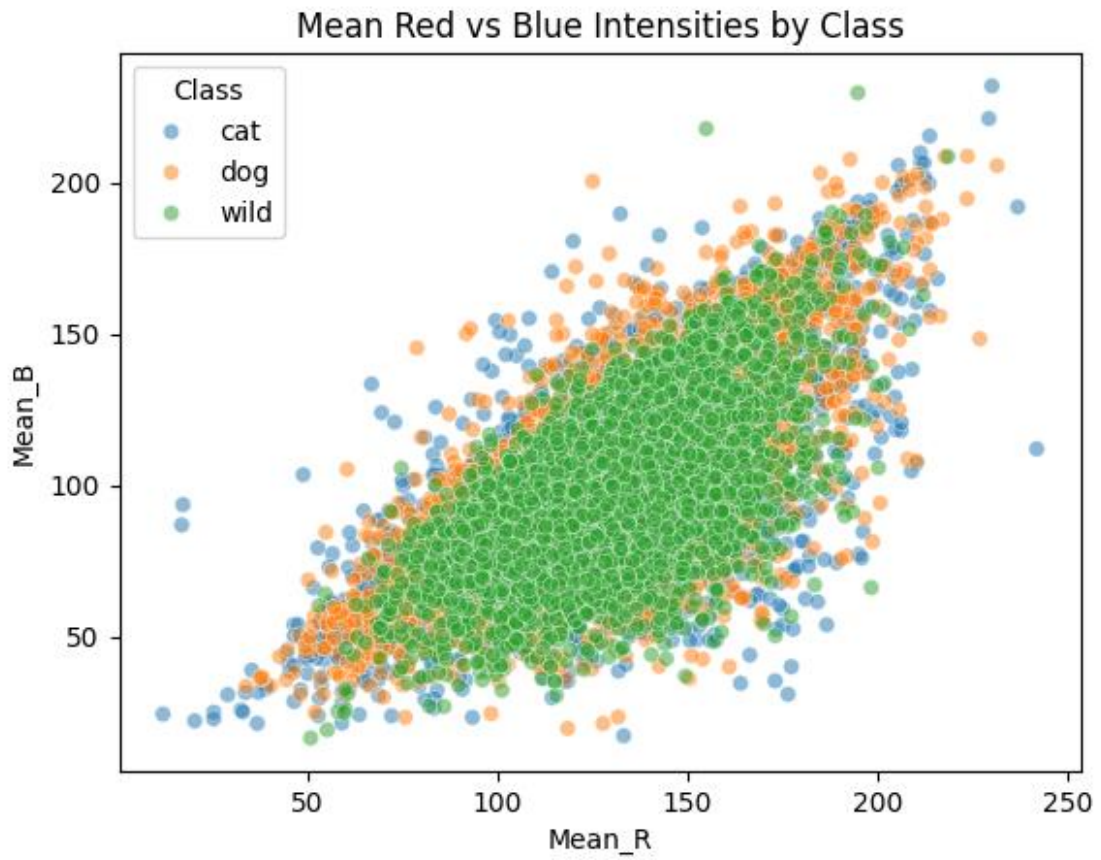


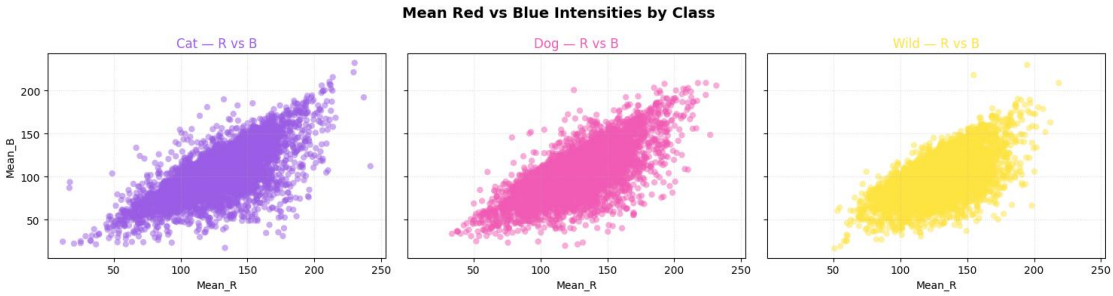
Moshiur Howlader



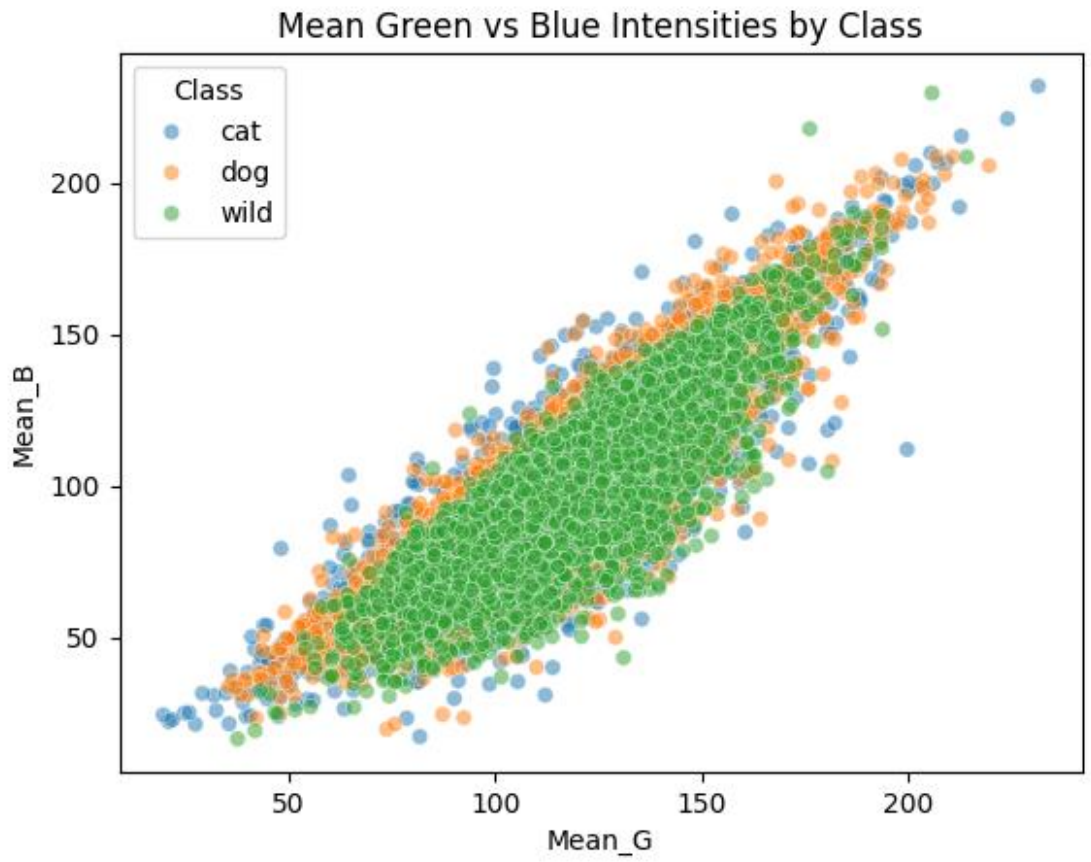


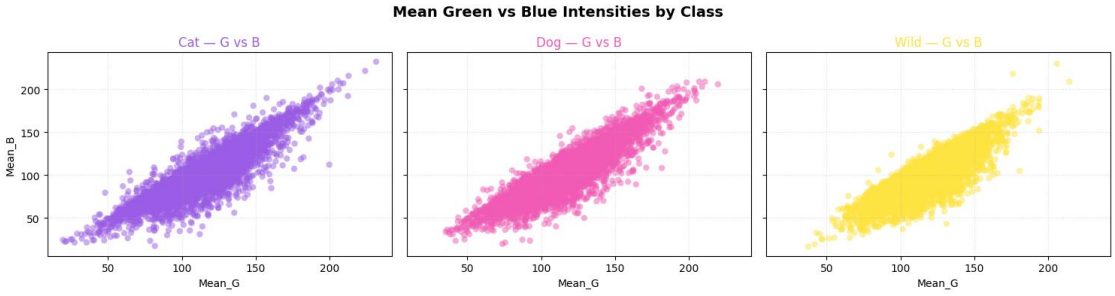
Moshiur Howlader

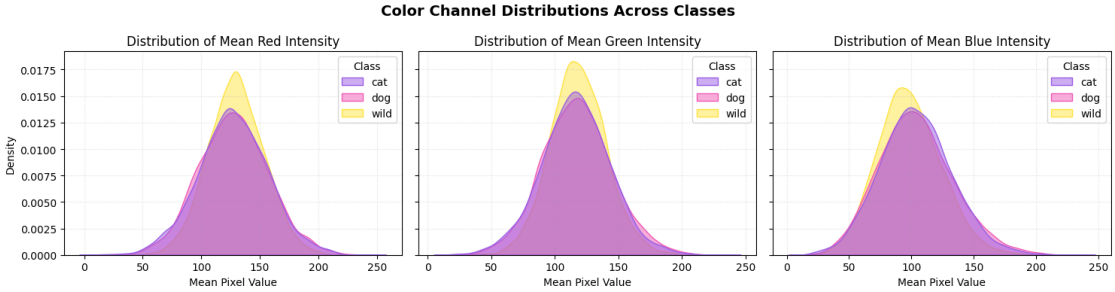




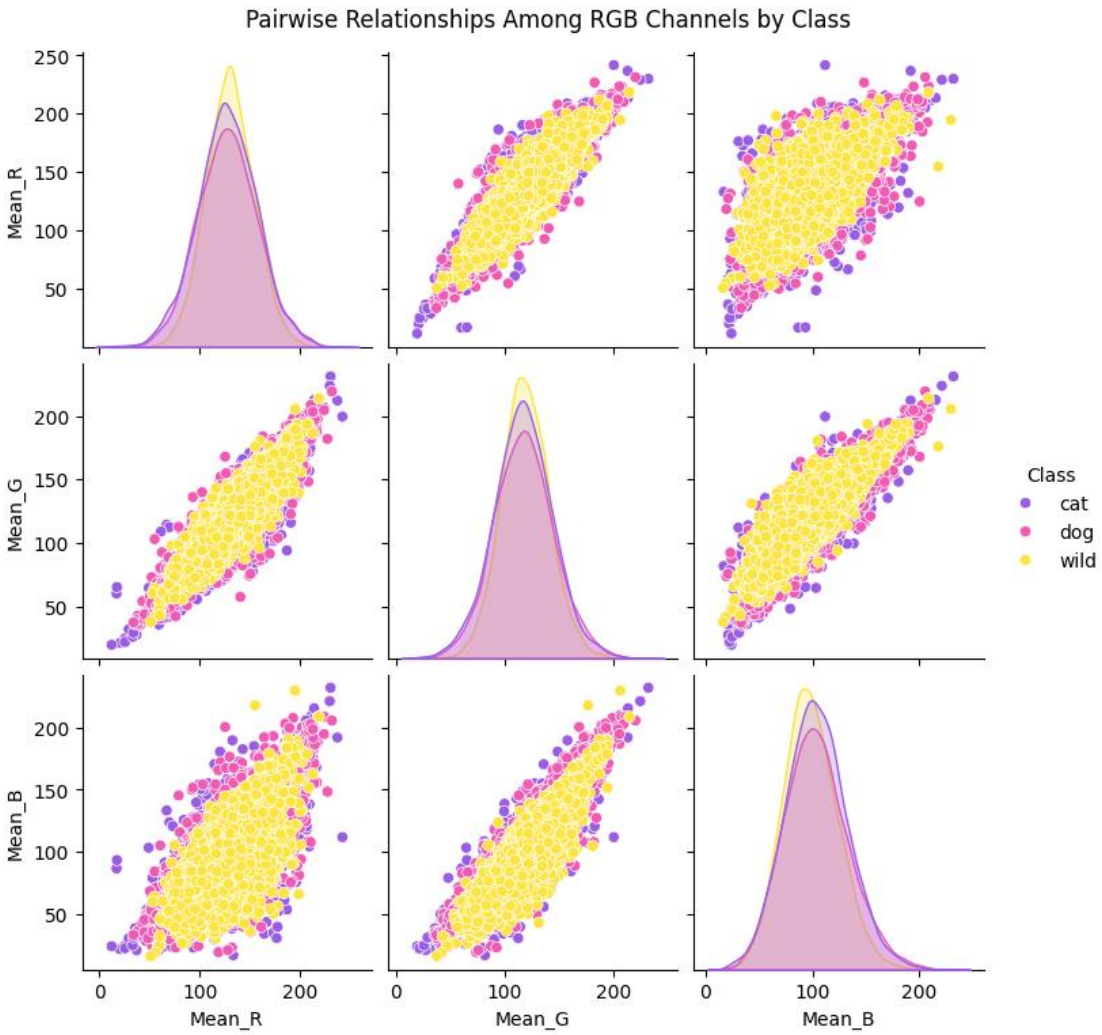
Moshiur Howlader

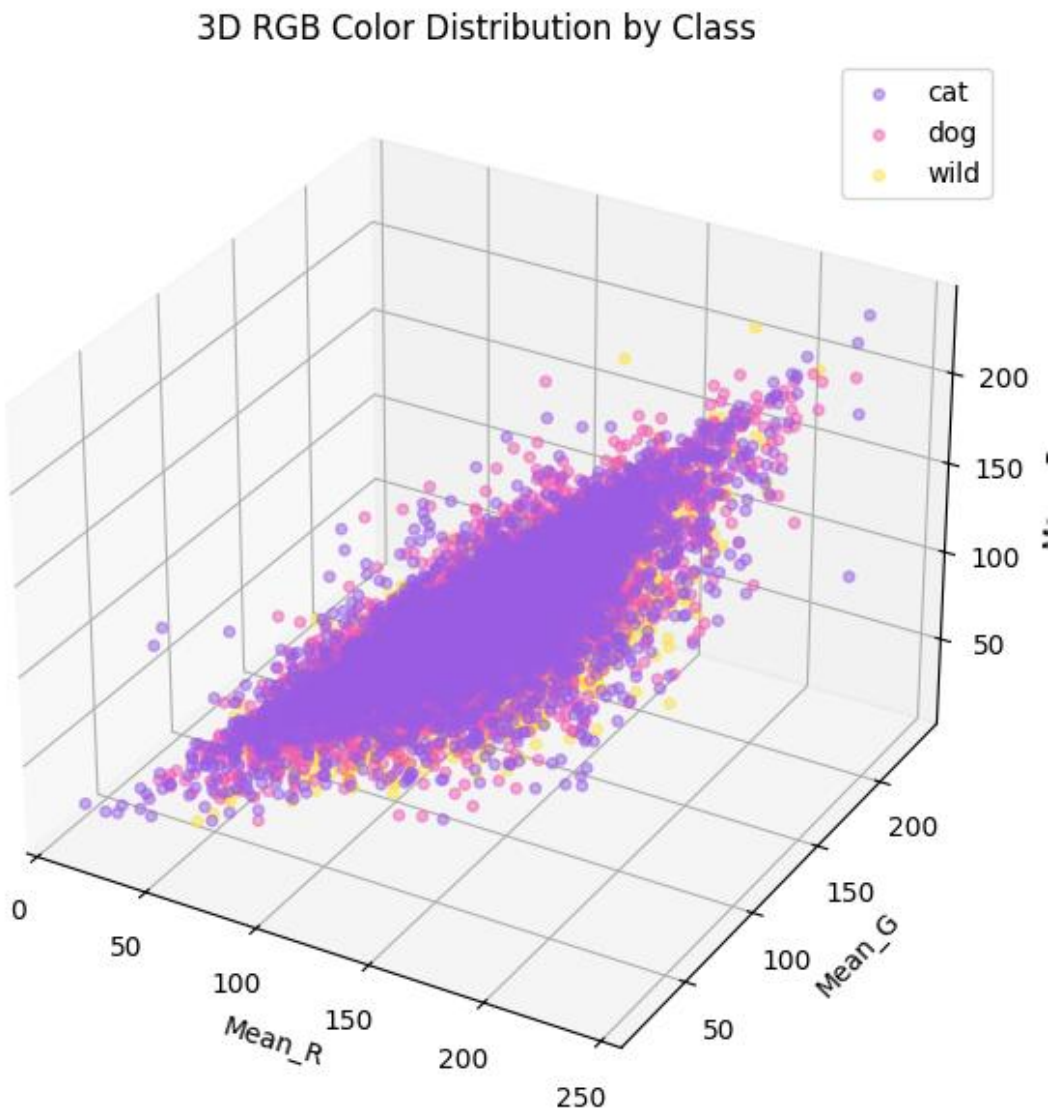






Moshiur Howlader





💡 **Explanation of results above:**

This section explores the relationships between the RGB color channels and visual intensity patterns across the AFHQ dataset.

- **Color Channel Correlation Matrix:**
  The heatmap shows that the Red and Green channels are strongly correlated (**r ≈ 0.89**), while both have strong negative correlations with Blue (**r ≈ –0.99 / –0.83**).
  In simpler terms, when an image has more red and green (warmer tones), it tends to have less blue — which is pretty typical for natural lighting and animal fur.

- **Boxplot (Outlier Check):**
  Each channel has a steady median intensity around **120–130**, with only a few extreme outliers.

That means most images are evenly exposed — not too bright, not too dark — and the dataset does not have color balance issues or washed-out samples.

- **Pairwise Scatterplots (R vs G, R vs B, G vs B):**
  The dense diagonal lines in the scatterplots show strong linear relationships between all color channels.
  When looking by class, the color distributions are quite similar overall — though wild animals lean a bit more blue, and cats and dogs are slightly warmer.
  It's a nice sign of consistent color calibration with just small, natural variations between species.

- **Channel-wise Density Distributions:**
  The overlapping, bell-shaped curves for Cat, Dog, and Wild suggest that pixel intensities follow a roughly normal distribution centered around **120 ± 25**.
  While all three classes share similar distributions, the **wild** category shows slightly higher density peaks across R, G, and B channels — indicating that wild animal images tend to be a bit brighter and more saturated overall.
  The strong overlap across classes still confirms balanced exposure and consistent color tone throughout the dataset.

- **Pairplot Summary:**
  The pairplot basically combines all those relationships in one grid, and it shows the same story — smooth color gradients and nearly linear trends between R, G, and B.

- **3D RGB Scatter Plot:**
  In 3D space, all three animal groups sit in overlapping regions, with only slight shifts in tone.
  That shows class differences are more about subtle textures and patterns rather than drastic color changes — exactly what you'd hope to see before running unsupervised models.

Overall Interpretation

The EDA shows that:

- Red and Green channels are strongly correlated, while Blue varies inversely — reflecting the natural balance between warm and cool tones in animal images.

- Lighting and exposure are generally consistent across classes, with wild images appearing slightly brighter on average.
- There are no major color outliers or illumination issues.

Together, these findings confirm that **AFHQ's color features are coherent, balanced, and well-suited for unsupervised analysis** such as clustering or feature extraction.

3.5 Data Quality and Cleaning

This section is short since it is known the data quality of the **AFHQ** is high quality.

```
import os

total = len(labels_train)
valid = sum([img is not None for img in data])
print(f"Valid images: {valid}/{total}")
```

```
Valid images: 14630/14630
```

💡 **Explanation of code above:**

As shown in the previous plots and verified by the code above, the dataset is clean, complete, and ready for use.

3.6 Transformations and Normalization

Since the RGB channel values are already on the same scale (0–255) and exhibit similar distributions across classes, no additional pixel level transformation is required at this stage.
However, we can normalize the dataset **in place** for models that expect input values within the [0, 1] range.
Maintaining a separate normalized copy of the dataset is prohibitively expensive (`MemoryError: Unable to allocate 42.9 GiB for an array with shape (14630, 512, 512, 3) and data type float32`), since floating-point values are more memory-intensive than `uint8`.

To mitigate this, alternative approaches include:

- Downsampling images to `128×128` pixels
- Using grayscale versions of the images
- Normalizing data **on the fly** during model training or feature extraction
- Using **memory-mapped arrays** (`np.memmap`) to handle normalized data efficiently without fully loading it into memory

Normalization can be applied just before model input or feature extraction to ensure compatibility with most machine learning workflows.
For downstream unsupervised tasks such as PCA or clustering, standard normalization or min–max scaling may be applied to ensure feature comparability, but logarithmic or other

nonlinear transformations are unnecessary. Later in the notebook, the image dataset will be downsampled to `128×128` to enable memory-efficient processing and computationally feasible model training, especially since some models require the use of `float32` datatypes.

```
# data_norm = data.astype(np.float32)
# np.divide(data_norm, 255.0, out=data_norm)

# print(data_norm.min(), data_norm.max())

# ---------------------------------------------------------------------------
# MemoryError                               Traceback (most recent call last)
# Cell In[10], line 1
# ----> 1 data_norm = data.astype(np.float32)
#       2 np.divide(data_norm, 255.0, out=data_norm)
#       4 print(data_norm.min(), data_norm.max())

# MemoryError: Unable to allocate 42.9 GiB for an array with shape (14630, 51
2, 512, 3) and data type float32
```

Although all three color channels (R, G, and B) have similar ranges and balanced distributions—making additional transformations such as log scaling unnecessary—it is still useful to apply **edge detection** to reduce the image data to its most informative structural features (i.e., object boundaries, shapes, and textures).

💡 **Explanation of code below:**
This code outputs the original RGB, as well as the three common edge detection techniques side by side, highlighting different visual details from the same image.

- **Canny:** Detects the strongest edges and outlines (like the whiskers and eyes). It gives a clean, black-and-white look that focuses only on major edges.
- **Sobel:** Captures softer gradients and textures. You can still see the fur and fine details, but with smoother transitions.
- **Laplacian:** Picks up very subtle edges and changes in brightness. It can look noisy, but helps reveal faint details that other filters might miss.

Overall, each method captures different kinds of information — structure, texture, and fine detail — which can be useful for feature extraction or unsupervised learning later on.

Sources:

- **Edge Detection using OpenCV (Official Blog)**
- **Edge Detection - LearnOpenCV**

```
data_edges_canny = []
data_edges_sobel = []
data_edges_laplacian = []
missing = 0

print("Generating edge-detected datasets (Canny, Sobel, Laplacian)...")

# Using float32 here provides negligible improvement in quality compared to i
nteger precision.
# edge_detection_output_int_versio.png vs edge_detection_full_float_version.p
ng
for cls in classes:
    folder = os.path.join(DATA_DIR, cls)
    for fname in tqdm(os.listdir(folder), desc=f"Processing {cls}"):
        fpath = os.path.join(folder, fname)
        img = cv2.imread(fpath)
        if img is None:
            missing += 1
            continue

        # Convert to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # --- Canny Edge Detection ---
        edges_canny = cv2.Canny(gray, threshold1=100, threshold2=200)

        # --- Sobel Edge Detection ---
        sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
        sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
        sobel = cv2.magnitude(sobelx, sobely)
        sobel = np.uint8(np.clip(sobel, 0, 255))

        # --- Laplacian Edge Detection ---
        laplacian = cv2.Laplacian(gray, cv2.CV_64F)
        laplacian = np.uint8(np.clip(np.abs(laplacian), 0, 255))

        # Append all
        data_edges_canny.append(edges_canny)
        data_edges_sobel.append(sobel)
        data_edges_laplacian.append(laplacian)

# Convert all to numpy arrays (uint8)
data_edges_canny = np.array(data_edges_canny, dtype=np.uint8)
data_edges_sobel = np.array(data_edges_sobel, dtype=np.uint8)
data_edges_laplacian = np.array(data_edges_laplacian, dtype=np.uint8)

print("☑ Canny shape:", data_edges_canny.shape)
print("☑ Sobel shape:", data_edges_sobel.shape)
```

```
print("☑ Laplacian shape:", data_edges_laplacian.shape)
print("Approx. total memory use (GB):",
      round((data_edges_canny.nbytes +
             data_edges_sobel.nbytes +
             data_edges_laplacian.nbytes) / (1024**3), 2))
print(f"⚠ Missing or unreadable files: {missing}")

# === Save Each Dataset Asynchronously ===
print("\n💾 Starting background saves...")

save_thread_canny = save_numpy_array_async(data_edges_canny, "AFHQ_edges_cann
y")
save_thread_sobel = save_numpy_array_async(data_edges_sobel, "AFHQ_edges_sobe
l")
save_thread_laplacian = save_numpy_array_async(data_edges_laplacian, "AFHQ_ed
ges_laplacian")

# Pick an index (0 = first image, or any number < len(data))
idx = 0

# Retrieve original and edge-detected versions
img_original = data[idx]
img_canny = data_edges_canny[idx]
img_sobel = data_edges_sobel[idx]
img_laplacian = data_edges_laplacian[idx]

# Plot all side-by-side
fig, axes = plt.subplots(1, 4, figsize=(16, 5))
axes[0].imshow(img_original)
axes[0].set_title("Original RGB Image")
axes[1].imshow(img_canny, cmap="gray")
axes[1].set_title("Canny Edge Detection")
axes[2].imshow(img_sobel, cmap="gray")
axes[2].set_title("Sobel Edge Detection")
axes[3].imshow(img_laplacian, cmap="gray")
axes[3].set_title("Laplacian Edge Detection")

for ax in axes:
    ax.axis("off")

plt.suptitle("Comparison of Edge Detection Methods", fontsize=14, weight="bol
d")
plt.tight_layout()
plt.show()

print("\n📉 Downsampling datasets to 128×128 for PCA and clustering...")

TARGET_SIZE = (128, 128)
```

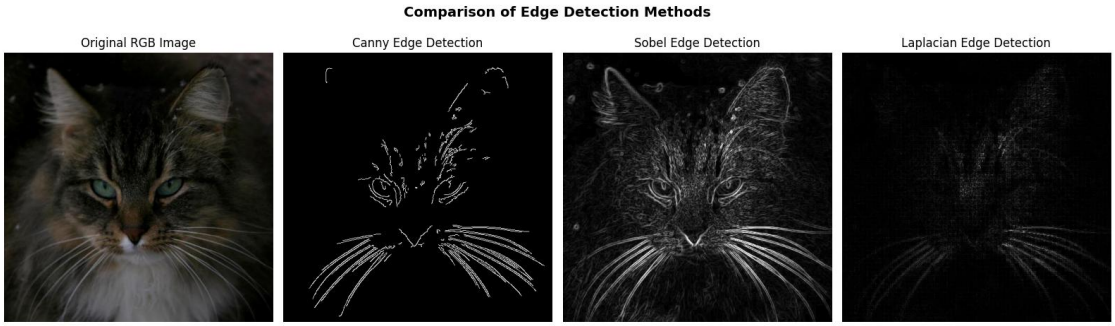
```
def resize_batch(batch, is_rgb=False):
    """Resize batch of images to 128x128.
       RGB -> resize each (h,w,3)
       Edge maps -> resize each (h,w)
    """
    if is_rgb:
        return np.array([
            cv2.resize(img, TARGET_SIZE, interpolation=cv2.INTER_AREA)
            for img in batch
        ], dtype=np.uint8)
    else:
        return np.array([
            cv2.resize(img, TARGET_SIZE, interpolation=cv2.INTER_AREA)
            for img in batch
        ], dtype=np.uint8)

# ------------------------------------------------
# Create downsampled datasets with _small suffix
# ------------------------------------------------
print("Resizing RGB dataset...")
data_small = resize_batch(data, is_rgb=True)

print("Resizing Canny edges...")
data_edges_canny_small = resize_batch(data_edges_canny)

print("Resizing Sobel edges...")
data_edges_sobel_small = resize_batch(data_edges_sobel)

print("Resizing Laplacian edges...")
data_edges_laplacian_small = resize_batch(data_edges_laplacian)

# ------------------------------------------------
# Shapes of downsampled datasets
# ------------------------------------------------
print("\n New Shapes After Downsampling:")
print("RGB_small:", data_small.shape)
print("Canny_small:", data_edges_canny_small.shape)
print("Sobel_small:", data_edges_sobel_small.shape)
print("Laplacian_small:", data_edges_laplacian_small.shape)

# Memory footprint
total_gb_small = round(
    (data_small.nbytes +
     data_edges_canny_small.nbytes +
     data_edges_sobel_small.nbytes +
     data_edges_laplacian_small.nbytes) / (1024**3),
    2
)
```

```
print(f"\n💾 Total Memory (Downsampled Sets): {total_gb_small} GB")
print("👍 Ready for PCA / clustering!")

# ------------------------------------------------
# Plot first image from SMALL datasets for comparison
# ------------------------------------------------
idx = 0  # choose any image index

img_rgb_small = data_small[idx]
img_canny_small = data_edges_canny_small[idx]
img_sobel_small = data_edges_sobel_small[idx]
img_laplacian_small = data_edges_laplacian_small[idx]

fig, axes = plt.subplots(1, 4, figsize=(16, 5))

axes[0].imshow(img_rgb_small)
axes[0].set_title("Downsampled RGB (128×128)")

axes[1].imshow(img_canny_small, cmap="gray")
axes[1].set_title("Canny (128×128)")

axes[2].imshow(img_sobel_small, cmap="gray")
axes[2].set_title("Sobel (128×128)")

axes[3].imshow(img_laplacian_small, cmap="gray")
axes[3].set_title("Laplacian (128×128)")

for ax in axes:
    ax.axis("off")

plt.suptitle("Comparison of Downsampled Edge Representations (128×128)",
             fontsize=14, weight="bold")

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

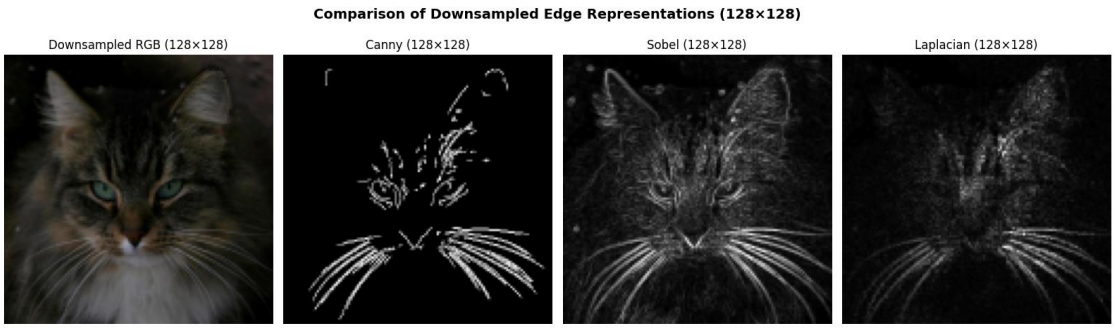
```
Generating edge-detected datasets (Canny, Sobel, Laplacian)...

Processing cat: 100%|██████████| 5153/5153 [00:27<00:00, 185.29it/s]
Processing dog: 100%|██████████| 4739/4739 [00:27<00:00, 170.04it/s]
Processing wild: 100%|██████████| 4738/4738 [00:32<00:00, 145.96it/s]

✅ Canny shape: (14630, 512, 512)
✅ Sobel shape: (14630, 512, 512)
✅ Laplacian shape: (14630, 512, 512)
Approx. total memory use (GB): 10.72
⚠️ Missing or unreadable files: 0
```

Moshiur Howlader

```
💾 Starting background saves...
⚙️ [Async Save Started] Saving AFHQ_edges_canny.pkl (3.57 GB) in backgroun
d...
⚙️ [Async Save Started] Saving AFHQ_edges_sobel.pkl (3.57 GB) in backgroun
d...
⚙️ [Async Save Started] Saving AFHQ_edges_laplacian.pkl (3.57 GB) in backgro
und...
```




```
📉 Downsampling datasets to 128×128 for PCA and clustering...
Resizing RGB dataset...
Resizing Canny edges...
Resizing Sobel edges...
Resizing Laplacian edges...

 New Shapes After Downsampling:
RGB_small: (14630, 128, 128, 3)
Canny_small: (14630, 128, 128)
Sobel_small: (14630, 128, 128)
Laplacian_small: (14630, 128, 128)

💾 Total Memory (Downsampled Sets): 1.34 GB
👍 Ready for PCA / clustering!
```




```
✅ [Async Save Complete] AFHQ_edges_canny.pkl (3.57 GB) saved to d:\Document
s\GitHub\csca5632-final-project\data\AFHQ_edges_canny.pkl
✅ [Async Save Complete] AFHQ_edges_laplacian.pkl (3.57 GB) saved to d:\Docu
```

```
ments\GitHub\csca5632-final-project\data\AFHQ_edges_laplacian.pkl
✅ [Async Save Complete] AFHQ_edges_sobel.pkl (3.57 GB) saved to d:\Document
s\GitHub\csca5632-final-project\data\AFHQ_edges_sobel.pkl
```

💡 **Explanation of output above:**
As one can visually see, downsampling the images from 512×512 to 128×128 preserves the essential structural information—such as shapes, contours, and major edges—while removing only fine-grained, high-frequency details that are not important for clustering. The reduced resolution is far more memory-efficient and makes PCA and unsupervised models computationally feasible without compromising the meaningful visual patterns in the data. Overall, the 128×128 representation remains highly informative while being significantly more ML-friendly.

3.6.1 Managing Memory - A Sidenote

Below is a code snippet for managing memory on systems with less **RAM** resources. The full dataset (`data`,`data_edges_canny`,`data_edges_sobel`,`data_edges_laplacian`) can be loaded on systems with 64 GB or more of RAM without manually freeing and loading datasets (although for this project, the downsampled 128x128 resolution will be used later to make model training computationally feasible and to save memory usage).

```
# # Safe to free memory after starting async saves – the background thread ke
eps its own reference ===
free_variable("data", globals())
free_variable("data_edges_canny", globals())
free_variable("data_edges_sobel", globals())
free_variable("data_edges_laplacian", globals())

# # Wait for all background threads to finish before reloading
# wait_for_threads(save_thread_rgb, save_thread_canny, save_thread_sobel, sav
e_thread_laplacian)

# # ☑ Safe to reload the saved datasets
# data = load_numpy_array("AFHQ_RGB_dataset")
# data_edges_canny = load_numpy_array("AFHQ_edges_canny")
# data_edges_sobel = load_numpy_array("AFHQ_edges_sobel")
# data_edges_laplacian = load_numpy_array("AFHQ_edges_laplacian")

# # 📊 Verify shapes, dtypes, and file sizes
# datasets = {
#     "RGB Dataset": ("AFHQ_RGB_dataset.pkl", data),
#     "Canny Edges": ("AFHQ_edges_canny.pkl", data_edges_canny),
#     "Sobel Edges": ("AFHQ_edges_sobel.pkl", data_edges_sobel),
#     "Laplacian Edges": ("AFHQ_edges_laplacian.pkl", data_edges_laplacian),
# }

# print("\n=== ☑ Dataset Verification Summary ===")
# for name, (fname, arr) in datasets.items():
```

```
#     fpath = os.path.join(VAR_DATA_DIR, fname)
#     fsize = get_file_size(fpath)
#     print(f"{name:<20} | Shape: {arr.shape} | Dtype: {arr.dtype} | File siz
e: {fsize} MB")

# # 💾 Print total memory footprint (in RAM)
# total_mem = memory_usage_gb(data, data_edges_canny, data_edges_sobel, data_
edges_laplacian)
# print(f"\n💡 Total Memory Usage (all datasets in memory): {total_mem} GB")
```

```
🧹 Deallocated variable: data
🧹 Deallocated variable: data_edges_canny
🧹 Deallocated variable: data_edges_sobel
🧹 Deallocated variable: data_edges_laplacian
```

Expected Output of the Full Code above (if clearing memory usage is required due to low memory system) is:

```
🧹 Deallocated variable: data
🧹 Deallocated variable: data_edges_canny
🧹 Deallocated variable: data_edges_sobel
🧹 Deallocated variable: data_edges_laplacian
✅ All background saves completed.
✅ Loaded AFHQ_RGB_dataset.pkl (10.72 GB) from d:\Documents\GitHub\csca5632-
final-project\data\AFHQ_RGB_dataset.pkl
✅ Loaded AFHQ_edges_canny.pkl (3.57 GB) from d:\Documents\GitHub\csca5632-f
inal-project\data\AFHQ_edges_canny.pkl
✅ Loaded AFHQ_edges_sobel.pkl (3.57 GB) from d:\Documents\GitHub\csca5632-f
inal-project\data\AFHQ_edges_sobel.pkl
✅ Loaded AFHQ_edges_laplacian.pkl (3.57 GB) from d:\Documents\GitHub\csca56
32-final-project\data\AFHQ_edges_laplacian.pkl

=== ✅ Dataset Verification Summary ===
RGB Dataset          | Shape: (14630, 512, 512, 3) | Dtype: uint8 | File siz
e: 8788.49 MB
Canny Edges          | Shape: (14630, 512, 512) | Dtype: uint8 | File size: 4
70.29 MB
Sobel Edges          | Shape: (14630, 512, 512) | Dtype: uint8 | File size: 3
349.2 MB
Laplacian Edges      | Shape: (14630, 512, 512) | Dtype: uint8 | File size: 2
978.1 MB

💡 Total Memory Usage (all datasets in memory): 21.43 GB
```

3.7 Feature Importance and Hypothesis

It is expected that clustering will be driven primarily by **texture**, **shape**, and **edge patterns**—such as fur structure, facial contours, and outline information—rather than by raw color values. Because **AFHQ** images vary widely in lighting, background, and color tone, color alone may not be a reliable feature for grouping similar animals. Edge filtering is therefore expected to reduce irrelevant color variation while preserving the underlying structural information, resulting in performance that is comparable to RGB, or only slightly lower (due to slight loss of information).

In particular, the assumption is that the large intensity transitions in RGB images (which define edges, boundaries, and key textures) would still be captured after applying edge filters. These structural cues were expected to provide enough latent patterns for clustering algorithms and CNN models to form meaningful groups, even when color information is removed.

3.8 Summary of EDA Findings

The exploratory data analysis (EDA) provided a detailed understanding of the **AFHQ dataset**, including its visual structure, data integrity, and readiness for downstream machine learning.

- **Data Quality and Balance:**
  All 14,630 images were successfully loaded and validated. The dataset is clean, balanced, and visually diverse across the three categories (Cat, Dog, Wild). No missing, corrupted, or poorly exposed samples were found.

- **Color Distributions:**
  The RGB channels showed strong correlations between Red and Green and an inverse relationship with Blue — typical of natural lighting and animal fur tones. Overlapping, bell-shaped distributions indicated well-balanced exposure across classes, confirming no hue or lighting bias.

- **Feature Relationships:**
  Pairwise scatterplots, 3D RGB projections, and correlation heatmaps showed smooth, near-linear relationships between color channels.
  These patterns suggest that color alone may not be a strong distinguishing factor among classes.

- **Transformations and Normalization:**
  Since all RGB channels share similar ranges (0–255) and distributions, global normalization or log transformations were deemed unnecessary.
  A full normalization attempt triggered a `MemoryError` (~43 GB allocation), highlighting the importance of efficient data handling.
  Alternatives such as on-the-fly normalization, downsampling, grayscale conversion, and memory-mapped arrays were discussed as practical options.

- **Edge-Based Representations:**
  Canny, Sobel, and Laplacian edge detection filters were applied to extract structural and textural features.
  Each method revealed complementary aspects of the same image — from sharp contours (Canny) to fine gradients (Sobel) and subtle intensity changes (Laplacian) — demonstrating the potential of structural features for unsupervised learning.

- **Memory Optimization:**
  Async save and load mechanisms were implemented to manage large datasets efficiently, reducing memory footprint while maintaining reproducibility.
  The full RGB and edge-based datasets occupy approximately **21 GB in RAM**, making them feasible for further processing on 64 GB+ RAM systems.

In summary:
The AFHQ dataset is high-quality, balanced, and ready for feature extraction and unsupervised learning.
Color statistics confirm consistency and neutrality, while edge detection experiments highlight the importance of **texture, shape, and contour features** over raw color values for downstream clustering or representation learning tasks.

---

4. Model Building and Training

Here the following models will be considered to build a system that can predict the image class:

4.1 Part A – "Classical" unsupervised on hand-crafted features

To establish a baseline, we first experiment with traditional unsupervised learning methods applied to several hand-crafted feature representations of the images. The four input types used are:

- **RGB pixel intensities**
- **Canny edge maps**
- **Sobel edge maps**
- **Laplacian edge maps**

These representations allow us to compare how different feature types influence cluster separability and whether simple edge-based features can retain most of the clustering performance over raw RGB data.

Feature Preprocessing Pipeline

Each feature set undergoes the same preprocessing steps:

1. **Flatten** each image into a 1D vector

2. **Normalize** all values to ([0, 1])

3. **Apply PCA** for dimensionality reduction
    - Various PCA component sizes (e.g., 1–100) are evaluated later in hyperparameter sweeps

This produces a compact representation that reduces noise and computational cost while preserving important structural information.

Unsupervised Models Evaluated

We evaluate the following clustering algorithms:

- **K-Means**
- **Gaussian Mixture Models (GMM)**
- **DBSCAN**
- **Agglomerative (Hierarchical) Clustering**

These algorithms represent several major families of clustering techniques: centroid-based, probabilistic, density-based, and hierarchical.

How Each Unsupervised Algorithm Works (High-Level Overview)

- **K-Means** partitions data into (k) clusters by iteratively assigning points to the nearest centroid. It assumes spherical, equally sized clusters and relies entirely on Euclidean distance.

- **Gaussian Mixture Models (GMM)** model each cluster as a Gaussian distribution with its own mean and covariance. Unlike K-Means, GMM can learn elliptical and overlapping cluster shapes, giving it more flexibility.

- **DBSCAN** groups points based on local density. It discovers arbitrary-shaped clusters but struggles in high-dimensional spaces where distances become less meaningful and density differences flatten out.

- **Agglomerative Clustering** builds a hierarchy by repeatedly merging the closest clusters according to a linkage rule (e.g., Ward). It can capture multi-scale structure but is sensitive to distance metrics and noise.

Why These Algorithms Struggle With Image Data

Clustering high-dimensional image data is inherently difficult. Even after PCA, the embeddings remain complex and nonlinear, and classic distance-based algorithms were not designed for this type of structure:

- **Distance concentration:** In high dimensions, Euclidean distances tend to become similar for all points, making it difficult to form meaningful clusters.
- **Loss of semantic meaning:** Flattened images and PCA projections preserve variance, not semantic concepts such as “cat vs dog.”
- **No hierarchical feature learning:** Classical clustering has no mechanism to learn patterns such as fur texture, ear shape, snout geometry, or facial structure.

This means all methods operate on features that have some structure or are statistically coherent but semantically weak or lack real semantic information, which limits the models' performance.

Algorithm-Specific Strengths and Limitations

- **K-Means** performs poorly because it assumes spherical clusters and relies entirely on Euclidean distance, which is not meaningful in PCA-transformed image space.
- **Gaussian Mixture Models (GMM)** perform better because they allow each cluster to learn its own covariance structure. This enables GMM to capture elongated or overlapping cluster shapes that occur naturally in image embeddings.
- **DBSCAN** fails in all but very low PCA dimensions because density estimation breaks down in high-dimensional spaces; most points appear equally dense, causing DBSCAN to either collapse into a single cluster or mark most points as noise.
- **Agglomerative Clustering** sometimes performs moderately well because hierarchical merging can capture coarse structure. However, once clusters are merged, they cannot be undone, making the method sensitive to early mistakes.

These limitations collectively explain why the performance differences between methods emerge so clearly in the results that follow.

Evaluation Procedure

Here the evaluation procedure used for the unsupervised learning algorithms will be outlined. For every combination of **feature type × clustering algorithm**, the following steps are performed:

1. **Fit the model on the training set** (`X_train_pca`)
2. **Evaluate clustering structure using external metrics:**
    - **Adjusted Rand Index (ARI)**
    - **Normalized Mutual Information (NMI)**
3. **Convert clusters to class predictions**
    - A majority-vote cluster-to-label mapping is computed on the training set

4. **Compute training accuracy** using this mapping
5. **Evaluate on the validation set**
    - Apply PCA transform using the *train* PCA model
    - Predict cluster labels
    - Apply the *same* train-derived majority-vote mapping
    - Compute **validation accuracy**

This ensures that all models are assessed consistently and that performance on unseen images is evaluated fairly.

The following code loads and preprocesses the validation dataset by resizing images, generating Canny/Sobel/Laplacian edge maps, and organizing everything into NumPy arrays for downstream evaluation:

```
print("📂 Loading validation set from:", VAL_DIR)

classes = os.listdir(VAL_DIR)

TARGET_SIZE = (128, 128)

data_small_val = []
data_edges_canny_small_val = []
data_edges_sobel_small_val = []
data_edges_laplacian_small_val = []

labels_val = []
filenames_val = []

missing = 0

for cls in classes:
    folder = os.path.join(VAL_DIR, cls)
    for fname in tqdm(os.listdir(folder), desc=f"VAL — {cls}"):

        fpath = os.path.join(folder, fname)
        img = cv2.imread(fpath)
        if img is None:
            print(f"⚠️ Skipping corrupted val image: {fname}")
            missing += 1
            continue

        # Convert BGR -> RGB
```

```
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Resize to 128×128 (same as training)
        img_rgb_small = cv2.resize(img_rgb, TARGET_SIZE, interpolation=cv2.IN
TER_AREA)

        # Store RGB
        data_small_val.append(img_rgb_small)
        labels_val.append(cls)
        filenames_val.append(fname)

        # Convert to grayscale for edge detection
        gray = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY)

        # --- Canny ---
        edges_canny = cv2.Canny(gray, 100, 200)

        # --- Sobel ---
        sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
        sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)
        sobel = cv2.magnitude(sobelx, sobely)
        sobel = np.uint8(np.clip(sobel, 0, 255))

        # --- Laplacian ---
        laplacian = cv2.Laplacian(gray, cv2.CV_64F)
        laplacian = np.uint8(np.clip(np.abs(laplacian), 0, 255))

        # Resize edge maps as well
        edges_canny_small = cv2.resize(edges_canny, TARGET_SIZE, interpolatio
n=cv2.INTER_AREA)
        sobel_small = cv2.resize(sobel, TARGET_SIZE, interpolation=cv2.INTER_
AREA)
        laplacian_small = cv2.resize(laplacian, TARGET_SIZE, interpolation=cv
2.INTER_AREA)

        data_edges_canny_small_val.append(edges_canny_small)
        data_edges_sobel_small_val.append(sobel_small)
        data_edges_laplacian_small_val.append(laplacian_small)

# Convert lists -> numpy arrays
data_small_val = np.array(data_small_val, dtype=np.uint8)
data_edges_canny_small_val = np.array(data_edges_canny_small_val, dtype=np.ui
nt8)
data_edges_sobel_small_val = np.array(data_edges_sobel_small_val, dtype=np.ui
nt8)
data_edges_laplacian_small_val = np.array(data_edges_laplacian_small_val, dty
pe=np.uint8)
```

```
labels_val = np.array(labels_val)
filenames_val = np.array(filenames_val)

print("\n☑ Finished loading VAL dataset!")
print("RGB val shape:", data_small_val.shape)
print("Canny val shape:", data_edges_canny_small_val.shape)
print("Sobel val shape:", data_edges_sobel_small_val.shape)
print("Laplacian val shape:", data_edges_laplacian_small_val.shape)
print("Labels val shape:", labels_val.shape)
print(f"Missing val files: {missing}")
```

```
📂 Loading validation set from: d:\Documents\GitHub\csca5632-final-project\d
ata\animal-faces\afhq\val

VAL — cat: 100%|██████████| 500/500 [00:06<00:00, 80.56it/s]
VAL — dog: 100%|██████████| 500/500 [00:05<00:00, 88.17it/s]
VAL — wild: 100%|██████████| 500/500 [00:05<00:00, 86.12it/s]


✅ Finished loading VAL dataset!
RGB val shape: (1500, 128, 128, 3)
Canny val shape: (1500, 128, 128)
Sobel val shape: (1500, 128, 128)
Laplacian val shape: (1500, 128, 128)
Labels val shape: (1500,)
Missing val files: 0
```

The following code flattens and normalizes all feature sets, applies PCA, runs multiple clustering algorithms, evaluates them on both train and validation data, and compiles the results into a summary table:

```
# import numpy as np
# from sklearn.decomposition import PCA
# from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
# from sklearn.mixture import GaussianMixture
# from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_sco
re
# import matplotlib.pyplot as plt
# from sklearn.neighbors import NearestCentroid

# # ======================================================
# # 1) PREPROCESSING = Flatten + Normalize
# # ======================================================

# def flatten_and_normalize(arr):
#     """Flatten to (N, D) and normalize to float32 [0, 1]."""
#     return arr.reshape(len(arr), -1).astype(np.float32) / 255.0
```

```
# print("Flattening & Normalizing input feature sets...")

# # TRAIN sets
# X_rgb = flatten_and_normalize(data_small)
# X_canny = flatten_and_normalize(data_edges_canny_small)
# X_sobel = flatten_and_normalize(data_edges_sobel_small)
# X_laplacian = flatten_and_normalize(data_edges_laplacian_small)

# # VAL sets (NEW)
# X_rgb_val = flatten_and_normalize(data_small_val)
# X_canny_val = flatten_and_normalize(data_edges_canny_small_val)
# X_sobel_val = flatten_and_normalize(data_edges_sobel_small_val)
# X_laplacian_val = flatten_and_normalize(data_edges_laplacian_small_val)

# print("Shapes:")
# print("RGB:", X_rgb.shape)
# print("Canny:", X_canny.shape)
# print("Sobel:", X_sobel.shape)
# print("Laplacian:", X_laplacian.shape)

# # Store in dictionary for experiment loop
# feature_sets = {
#     "rgb": X_rgb,
#     "canny": X_canny,
#     "sobel": X_sobel,
#     "laplacian": X_laplacian
# }

# feature_sets_val = {
#     "rgb": X_rgb_val,
#     "canny": X_canny_val,
#     "sobel": X_sobel_val,
#     "laplacian": X_laplacian_val
# }

# # PCA dimension
# PCA_COMPONENTS = 100


# # ============================================================
# # 2) Helper Functions
# # ============================================================

# def run_pca(X):
#     """Fit PCA and transform input features."""
#     pca = PCA(n_components=PCA_COMPONENTS, random_state=42)
#     X_pca = pca.fit_transform(X)
#     return X_pca, pca
```

```
# # Predict for DBSCAN/Agglomerative (fallback method)
# def predict_with_centroids(train_X, train_clusters, val_X):
#     clf = NearestCentroid()
#     clf.fit(train_X, train_clusters)
#     return clf.predict(val_X)


# def cluster_with_kmeans(X):
#     model = KMeans(n_clusters=3, random_state=42)
#     labels = model.fit_predict(X)
#     return labels, model


# def cluster_with_gmm(X):
#     model = GaussianMixture(n_components=3, covariance_type='diag', random_
state=42)
#     labels = model.fit_predict(X)
#     return labels, model


# def cluster_with_dbscan(X):
#     # You can tune eps/min_samples here
#     model = DBSCAN(eps=2.5, min_samples=25)
#     labels = model.fit_predict(X)
#     return labels, model


# def cluster_with_agglomerative(X):
#     model = AgglomerativeClustering(n_clusters=3, linkage="ward")
#     labels = model.fit_predict(X)
#     return labels, model


# def evaluate_clustering(true_labels, pred_labels):
#     """Compute ARI and NMI."""
#     ari = adjusted_rand_score(true_labels, pred_labels)
#     nmi = normalized_mutual_info_score(true_labels, pred_labels)
#     return ari, nmi


# def map_clusters_to_labels(pred_clusters, true_labels):
#     """
#     Majority-vote mapping: cluster -> real class.
#     Handles DBSCAN noise label (-1).
#     """
#     mapping = {}
#     for cluster_id in np.unique(pred_clusters):
```

```
#         if cluster_id == -1:
#             # DBSCAN noise: map to a dummy label
#             mapping[cluster_id] = "noise"
#             continue

#         idx = np.where(pred_clusters == cluster_id)[0]
#         cluster_true = true_labels[idx]

#         unique, counts = np.unique(cluster_true, return_counts=True)
#         majority = unique[np.argmax(counts)]
#         mapping[cluster_id] = majority

#     mapped = np.array([mapping[c] for c in pred_clusters])
#     accuracy = np.mean(mapped == true_labels)
#     return mapping, mapped, accuracy


# def plot_clusters(X_pca, labels, title):
#     """2D PCA scatter plot for visual inspection."""
#     pca2 = PCA(n_components=2)
#     X2 = pca2.fit_transform(X_pca)

#     plt.figure(figsize=(7,5))
#     plt.scatter(X2[:,0], X2[:,1], c=labels, s=5, cmap='viridis')
#     plt.title(title)
#     plt.show()


# # ========================================================
# # 3) Main Pipeline Loop
# # ========================================================

# clustering_algorithms = {
#     "kmeans": cluster_with_kmeans,
#     "gmm": cluster_with_gmm,
#     "dbscan": cluster_with_dbscan,
#     "agglomerative": cluster_with_agglomerative
# }

# results = []

# for feature_name, X in feature_sets.items():

#     print(f"\n==============================")
#     print(f"Working on feature: {feature_name}")
#     print(f"==============================")
```

```
#     # --- PCA (TRAIN) ---
#     X_pca, pca_model = run_pca(X)

#     # --- PCA (VAL) -- added ---
#     X_val = feature_sets_val[feature_name]
#     X_val_pca = pca_model.transform(X_val)

#     for algo_name, algo_fn in clustering_algorithms.items():

#         print(f"\n Algorithm: {algo_name} on {feature_name}")

#         # --- Clustering (TRAIN) ---
#         pred_clusters, model = algo_fn(X_pca)

#         # --- Train Metrics (ARI, NMI) ---
#         ari, nmi = evaluate_clustering(labels_train, pred_clusters)
#         mapping, mapped_preds, acc = map_clusters_to_labels(pred_clusters,
labels_train)

#         # ------------------------------------------------------------
#         # VAL predictions
#         # ------------------------------------------------------------
#         if algo_name in ["kmeans", "gmm"]:
#             val_clusters = model.predict(X_val_pca)
#         else:
#             # DBSCAN / Agglomerative: fallback prediction
#             unique_clusters = np.unique(pred_clusters)

#             if len(unique_clusters) < 2:
#                 # Only one cluster -> assign the same cluster to all val sa
mples
#                 val_clusters = np.full(len(X_val_pca), unique_clusters[0])
#             else:
#                 val_clusters = predict_with_centroids(X_pca, pred_clusters,
 X_val_pca)

#         # Apply TRAIN mapping to VAL clusters
#         val_preds = np.array([mapping[c] if c in mapping else "noise" for c
 in val_clusters])
#         val_acc = np.mean(val_preds == labels_val)

#         # --- Save results ---
#         results.append({
#             "features": feature_name,
#             "algorithm": algo_name,
#             "Train_ARI": ari,
#             "Train_NMI": nmi,
#             "Train_Accuracy": acc,
#             "Val_Accuracy": val_acc
```

```
#         })

#         # --- Print summary ---
#         print(f"    Mapping: {mapping}")
#         print(f"    ARI={ari:.4f}, NMI={nmi:.4f}, Accuracy={acc:.4f}")

#         # --- Optional visualization ---
#         # plot_clusters(X_pca, pred_clusters,
#         #               title=f"{algo_name.upper()} on {feature_name.upper
()} (PCA 2D)")


# # ============================================================
# # 4) Display Results Table
# # ============================================================

# import pandas as pd
# df_results = pd.DataFrame(results)
# display(df_results)
# df_results.to_csv("baseline_results.csv", index=False)

# ============================================================
# Run the Cached Table (takes about 1 minute, above code takes 5 minutes)
# ============================================================
df_results = pd.read_csv("baseline_results.csv")
display(df_results)
```

	features	algorithm	Train_ARI	Train_NMI	Train_Accuracy	Val_Accuracy
0	rgb	kmeans	0.009608	0.010389	0.373548	0.377333
1	rgb	gmm	0.046103	0.064986	0.438072	0.444000
2	rgb	dbscan	0.000000	0.000000	0.000000	0.000000
3	rgb	agglomerative	0.003666	0.003246	0.367669	
4	canny	kmeans	0.057255	0.059950	0.454067	
5	canny	gmm	0.021897	0.024237	0.401094	
6	canny	dbscan	0.000414	0.003171	0.007997	
7	canny	agglomerative	0.050825	0.055302	0.444839	
8	sobel	kmeans	0.076844	0.084139	0.474778	
9	sobel	gmm	0.009543	0.010980	0.392208	
10	sobel	dbscan	0.000000	0.000000	0.000000	
11	sobel	agglomerative	0.077683	0.083447	0.476418	
12	laplacian	kmeans	0.019761	0.024510	0.410731	
13	laplacian	gmm	0.009357	0.010026	0.387833	
14	laplacian	dbscan	0.008973	0.010082	0.085783	
15	laplacian	agglomerative	0.022832	0.024787	0.413124	

```
3       0.336667
4       0.444000
5       0.394000
6       0.146667
7       0.454000
8       0.470667
9       0.394000
10      0.000000
11      0.458000
12      0.416000
13      0.384667
14      0.204667
15      0.415333
```

Try tuning the hyper-parameters to see if there can be meaningful improvements:

```
# import numpy as np
# import pandas as pd
# from sklearn.decomposition import PCA
# from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
# from sklearn.mixture import GaussianMixture
# from sklearn.metrics import adjusted_rand_score, normalized_mutual_info_sco
re
# from sklearn.neighbors import NearestCentroid


# # ============================================================
# # Helper Functions
# # ============================================================

# def run_pca(X, n_components=100):
#     pca = PCA(n_components=n_components, random_state=42)
#     X_pca = pca.fit_transform(X)
#     return X_pca, pca


# def majority_map(pred_clusters, true_labels):
#     mapping = {}
#     for c in np.unique(pred_clusters):
#         if c == -1:  # DBSCAN noise
#             mapping[c] = "noise"
#             continue

#         idx = np.where(pred_clusters == c)[0]
#         votes = true_labels[idx]
#         unique, counts = np.unique(votes, return_counts=True)
#         mapping[c] = unique[np.argmax(counts)]

#     mapped = np.array([mapping[c] for c in pred_clusters])
```

```
#     acc = np.mean(mapped == true_labels)
#     return mapping, mapped, acc


# def evaluate(true, pred):
#     ari = adjusted_rand_score(true, pred)
#     nmi = normalized_mutual_info_score(true, pred)
#     return ari, nmi


# def centroid_predict(train_X, train_clusters, val_X):
#     """Fallback for DBSCAN / Agglomerative which have no predict()."""
#     unique_clusters = np.unique(train_clusters)

#     if len(unique_clusters) <= 1:
#         # Cannot build centroids — use majority class for all
#         return np.array([unique_clusters[0]] * len(val_X))

#     clf = NearestCentroid()
#     clf.fit(train_X, train_clusters)
#     return clf.predict(val_X)


# # ============================================================
# # Hyperparameter Grids
# # ============================================================

# PCA_LIST = [1, 10, 20, 50, 75, 100]

# kmeans_grid = {
#     "n_clusters": [2, 3, 4, 5]
# }

# gmm_grid = {
#     "n_components": [2, 3, 4, 5],
#     "covariance_type": ["diag", "full"]
# }

# dbscan_grid = {
#     "eps": [1.0, 1.5, 2.0, 2.5, 3.0],
#     "min_samples": [5, 10, 20, 30]
# }

# agglo_grid = {
#     "n_clusters": [2, 3, 4, 5],
#     "linkage": ["ward", "average", "complete"]
# }
```

```
# # ============================================================
# # Hyperparameter Sweep
# # ============================================================

# results = []

# for feature_name, X_train in feature_sets.items():

#     print(f"\n==============================")
#     print(f"Feature type: {feature_name}")
#     print("==============================")

#     # # --- PCA Train ---
#     # X_train_pca, pca_model = run_pca(X_train)

#     # # --- PCA Val ---
#     # X_val = feature_sets_val[feature_name]
#     # X_val_pca = pca_model.transform(X_val)

#     # --- PCA SWEEP ---
#     for pca_dim in PCA_LIST:

#         X_train_pca, pca_model = run_pca(X_train, n_components=pca_dim)
#         X_val = feature_sets_val[feature_name]
#         X_val_pca = pca_model.transform(X_val)

#         print(f" PCA = {pca_dim}")

#         # ============================================================
#         # K-MEANS
#         # ============================================================
#         for k in kmeans_grid["n_clusters"]:

#             model = KMeans(n_clusters=k, random_state=42)
#             train_clusters = model.fit_predict(X_train_pca)

#             # Train metrics
#             ari, nmi = evaluate(labels_train, train_clusters)
#             mapping, _, train_acc = majority_map(train_clusters, labels_tra
in)

#             # Val prediction
#             val_clusters = model.predict(X_val_pca)

#             # Apply train mapping
#             val_preds = np.array([mapping.get(c, "noise") for c in val_clus
ters])
```

```
#             val_acc = np.mean(val_preds == labels_val)

#             results.append({
#                 "features": feature_name,
#                 "algorithm": "kmeans",
#                 "PCA_dim": pca_dim,
#                 "params": f"k={k}",
#                 "Train_ARI": ari,
#                 "Train_NMI": nmi,
#                 "Train_Accuracy": train_acc,
#                 "Val_Accuracy": val_acc
#             })


#         # ============================================================
#         # GMM
#         # ============================================================
#         for k in gmm_grid["n_components"]:
#             for cov in gmm_grid["covariance_type"]:

#                 try:
#                     model = GaussianMixture(n_components=k, covariance_type
=cov, random_state=42)
#                     train_clusters = model.fit_predict(X_train_pca)
#                 except:
#                     continue

#                 # Train metrics
#                 ari, nmi = evaluate(labels_train, train_clusters)
#                 mapping, _, train_acc = majority_map(train_clusters, labels
_train)

#                 # Val
#                 val_clusters = model.predict(X_val_pca)
#                 val_preds = np.array([mapping.get(c, "noise") for c in val_
clusters])
#                 val_acc = np.mean(val_preds == labels_val)

#                 results.append({
#                     "features": feature_name,
#                     "algorithm": "gmm",
#                     "PCA_dim": pca_dim,
#                     "params": f"k={k}, cov={cov}",
#                     "Train_ARI": ari,
#                     "Train_NMI": nmi,
#                     "Train_Accuracy": train_acc,
#                     "Val_Accuracy": val_acc
#                 })
```

```
#         # ========================================================
#         # DBSCAN
#         # ========================================================
#         for eps in dbscan_grid["eps"]:
#             for ms in dbscan_grid["min_samples"]:

#                 model = DBSCAN(eps=eps, min_samples=ms)
#                 train_clusters = model.fit_predict(X_train_pca)

#                 # Train
#                 ari, nmi = evaluate(labels_train, train_clusters)
#                 mapping, _, train_acc = majority_map(train_clusters, labels
_train)

#                 # DBSCAN predict fallback
#                 val_clusters = centroid_predict(X_train_pca, train_cluster
s, X_val_pca)

#                 val_preds = np.array([mapping.get(c, "noise") for c in val_
clusters])
#                 val_acc = np.mean(val_preds == labels_val)

#                 results.append({
#                     "features": feature_name,
#                     "algorithm": "dbscan",
#                     "PCA_dim": pca_dim,
#                     "params": f"eps={eps}, min_samples={ms}",
#                     "Train_ARI": ari,
#                     "Train_NMI": nmi,
#                     "Train_Accuracy": train_acc,
#                     "Val_Accuracy": val_acc
#                 })


#         # ========================================================
#         # AGGLOMERATIVE
#         # ========================================================
#         for k in agglo_grid["n_clusters"]:
#             for link in agglo_grid["linkage"]:

#                 try:
#                     model = AgglomerativeClustering(n_clusters=k, linkage=l
ink)
#                     train_clusters = model.fit_predict(X_train_pca)
#                 except:
#                     continue
```

```
#                 # Train
#                 ari, nmi = evaluate(labels_train, train_clusters)
#                 mapping, _, train_acc = majority_map(train_clusters, labels
_train)

#                 # Predict fallback
#                 val_clusters = centroid_predict(X_train_pca, train_cluster
s, X_val_pca)

#                 val_preds = np.array([mapping.get(c, "noise") for c in val_
clusters])
#                 val_acc = np.mean(val_preds == labels_val)

#                 results.append({
#                     "features": feature_name,
#                     "algorithm": "agglomerative",
#                     "PCA_dim": pca_dim,
#                     "params": f"k={k}, linkage={link}",
#                     "Train_ARI": ari,
#                     "Train_NMI": nmi,
#                     "Train_Accuracy": train_acc,
#                     "Val_Accuracy": val_acc
#                 })



# # ============================================================
# # Output Table
# # ============================================================
# df_hyper = pd.DataFrame(results)
# df_hyper_sorted = df_hyper.sort_values("Val_Accuracy", ascending=False)

# pd.set_option('display.max_rows', None)
# pd.set_option('display.max_columns', None)
# pd.set_option('display.width', None)

# display(df_hyper_sorted)
# df_hyper_sorted.to_csv("hyperparameter_results.csv", index=False)



# ============================================================
# Run the Cached Table (hyper parameter sweep takes about 30 minutes)
# ============================================================
df_hyper_sorted = pd.read_csv("hyperparameter_results.csv")
display(df_hyper_sorted)
```

```
     features algorithm  PCA_dim                  params  Train_ARI  \
0         rgb       gmm      100           k=5, cov=full   0.266258
1         rgb       gmm       75           k=4, cov=full   0.283379
```

```
2         rgb       gmm       75            k=5, cov=full   0.278839
3         rgb       gmm      100            k=4, cov=full   0.311298
4         rgb       gmm       50            k=4, cov=full   0.218267
...       ...       ...      ...                      ...        ...
1051      rgb    dbscan       75  eps=2.5, min_samples=30   0.000000
1052      rgb    dbscan       75  eps=3.0, min_samples=10   0.000000
1053      rgb    dbscan       75   eps=3.0, min_samples=5   0.000000
1054      rgb    dbscan       75  eps=3.0, min_samples=30   0.000000
1055      rgb    dbscan       75  eps=3.0, min_samples=20   0.000000

      Train_NMI  Train_Accuracy  Val_Accuracy
0      0.300976        0.739234      0.750000
1      0.283965        0.708407      0.721333
2      0.291584        0.728776      0.719333
3      0.317562        0.717225      0.716667
4      0.208262        0.664593      0.678000
...         ...             ...           ...
1051   0.000000        0.000000      0.000000
1052   0.000000        0.000000      0.000000
1053   0.000000        0.000000      0.000000
1054   0.000000        0.000000      0.000000
1055   0.000000        0.000000      0.000000

[1056 rows x 8 columns]
```

Here a summary of the baseline results as well as the hyperparameter results for the unsupervised machine learning models are shown:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

df_base = pd.read_csv("baseline_results.csv")
df_hyper = pd.read_csv("hyperparameter_results.csv")

display(df_base.head())
display(df_hyper.head())
```

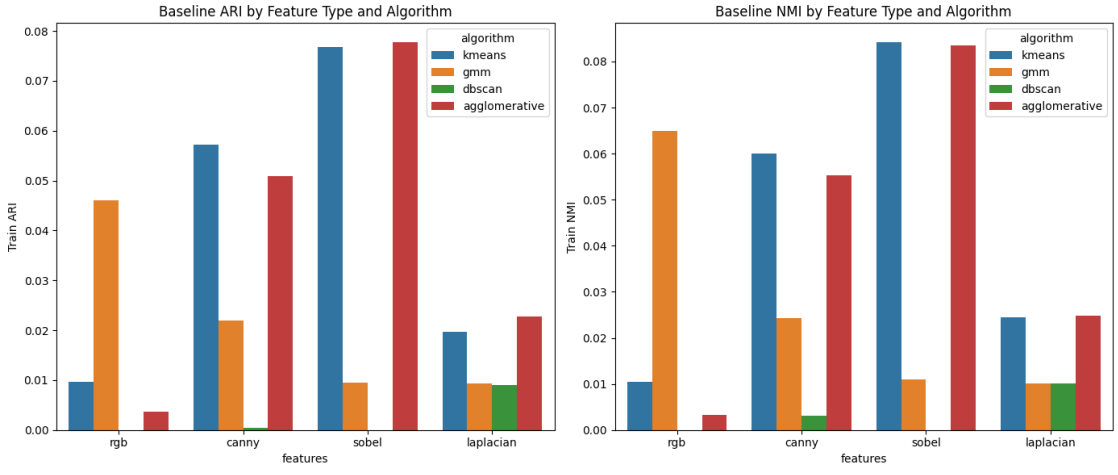
```
  features      algorithm  Train_ARI  Train_NMI  Train_Accuracy  Val_Accuracy
0      rgb         kmeans   0.009608   0.010389        0.373548      0.377333
1      rgb            gmm   0.046103   0.064986        0.438072      0.444000
2      rgb         dbscan   0.000000   0.000000        0.000000      0.000000
3      rgb  agglomerative   0.003666   0.003246        0.367669      0.336667
4    canny         kmeans   0.057255   0.059950        0.454067      0.444000

  features algorithm  PCA_dim         params  Train_ARI  Train_NMI  \
0      rgb       gmm      100  k=5, cov=full   0.266258   0.300976
1      rgb       gmm       75  k=4, cov=full   0.283379   0.283965
2      rgb       gmm       75  k=5, cov=full   0.278839   0.291584
3      rgb       gmm      100  k=4, cov=full   0.311298   0.317562
```

```
4      rgb       gmm       50  k=4, cov=full   0.218267   0.208262
```

```
   Train_Accuracy  Val_Accuracy
0        0.739234      0.750000
1        0.708407      0.721333
2        0.728776      0.719333
3        0.717225      0.716667
4        0.664593      0.678000
```

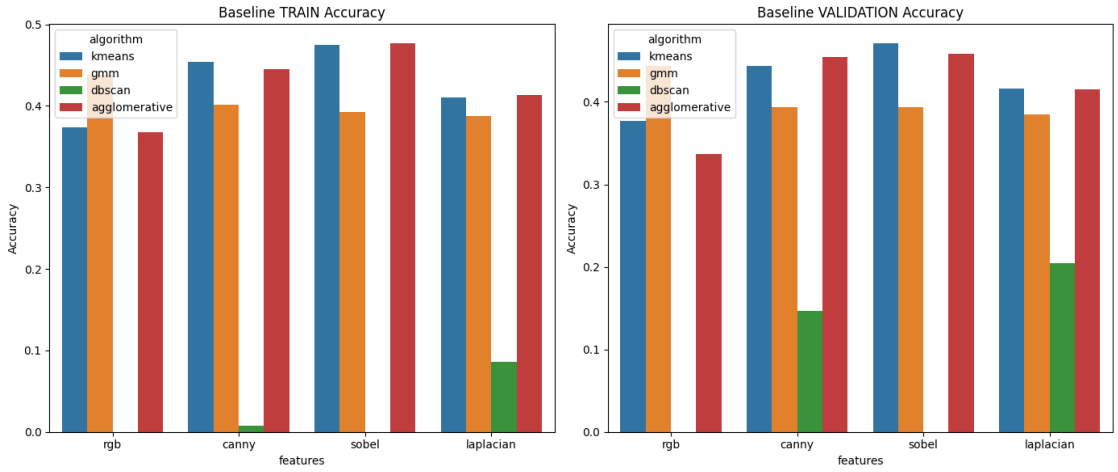
The code below visualizes **ARI** and **NMI** scores across different feature types and clustering algorithms to compare baseline performance.
ARI (Adjusted Rand Index) measures how closely the predicted clusters match the true labels (corrected for chance).
NMI (Normalized Mutual Information) measures how much information the predicted clusters share with the true labels.
Both metrics range from 0 to 1, with 1 indicating perfect clustering performance.

```
fig, axes = plt.subplots(1, 2, figsize=(14,6))

sns.barplot(data=df_base, x="features", y="Train_ARI", hue="algorithm", ax=ax
es[0])
axes[0].set_title("Baseline ARI by Feature Type and Algorithm")
axes[0].set_ylabel("Train ARI")

sns.barplot(data=df_base, x="features", y="Train_NMI", hue="algorithm", ax=ax
es[1])
axes[1].set_title("Baseline NMI by Feature Type and Algorithm")
axes[1].set_ylabel("Train NMI")

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



The plot below compares training and validation accuracy across feature types and clustering algorithms to assess baseline classification performance.

```
fig, axes = plt.subplots(1, 2, figsize=(14,6))

sns.barplot(data=df_base, x="features", y="Train_Accuracy", hue="algorithm",
ax=axes[0])
axes[0].set_title("Baseline TRAIN Accuracy")
axes[0].set_ylabel("Accuracy")

sns.barplot(data=df_base, x="features", y="Val_Accuracy", hue="algorithm", ax
=axes[1])
axes[1].set_title("Baseline VALIDATION Accuracy")
axes[1].set_ylabel("Accuracy")

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



The table output below lists the top 20 performing unsupervised ML models after performing the hyperparameter sweep:

```
df_best = df_hyper.sort_values("Val_Accuracy", ascending=False).head(20)
display(df_best)
```

	features	algorithm	PCA_dim	params	Train_ARI	Train_NMI	\
0	rgb	gmm	100	k=5, cov=full	0.266258	0.300976	
1	rgb	gmm	75	k=4, cov=full	0.283379	0.283965	
2	rgb	gmm	75	k=5, cov=full	0.278839	0.291584	
3	rgb	gmm	100	k=4, cov=full	0.311298	0.317562	
4	rgb	gmm	50	k=4, cov=full	0.218267	0.208262	
5	rgb	gmm	50	k=5, cov=full	0.186489	0.196610	
6	rgb	gmm	50	k=3, cov=full	0.216892	0.202403	
7	sobel	gmm	50	k=5, cov=full	0.155832	0.175304	
8	rgb	gmm	75	k=3, cov=full	0.201064	0.195907	
9	sobel	gmm	50	k=4, cov=full	0.161829	0.167520	
10	sobel	gmm	100	k=5, cov=full	0.152124	0.180200	
11	sobel	gmm	20	k=4, cov=full	0.143634	0.135816	
12	rgb	gmm	100	k=3, cov=full	0.190079	0.193874	

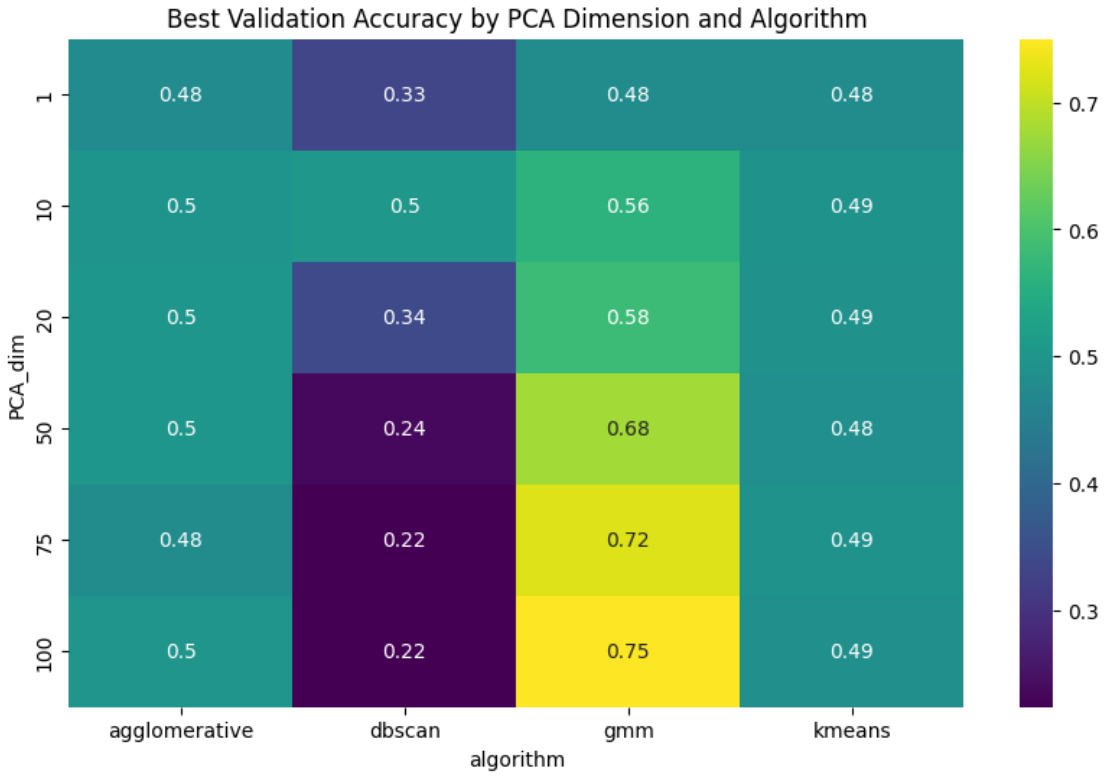
```
13    sobel       gmm       75  k=4, cov=full   0.159726   0.170100
14    sobel       gmm       20  k=5, cov=full   0.120424   0.120024
15    sobel       gmm      100  k=4, cov=full   0.163402   0.173691
16    sobel       gmm       10  k=4, cov=full   0.129012   0.121986
17      rgb       gmm       20  k=5, cov=full   0.118249   0.143925
18      rgb       gmm       75  k=4, cov=diag   0.104812   0.116340
19      rgb       gmm       75  k=5, cov=diag   0.081978   0.116170
```

```
    Train_Accuracy  Val_Accuracy
0         0.739234      0.750000
1         0.708407      0.721333
2         0.728776      0.719333
3         0.717225      0.716667
4         0.664593      0.678000
5         0.633083      0.636667
6         0.626931      0.628000
7         0.607382      0.602000
8         0.605742      0.600000
9         0.584074      0.588667
10        0.577649      0.587333
11        0.592686      0.582000
12        0.582638      0.576000
13        0.572864      0.572667
14        0.577649      0.566000
15        0.565414      0.562667
16        0.575120      0.561333
17        0.557690      0.558667
18        0.538893      0.555333
19        0.542584      0.553333
```

The heatmap output below shows how validation accuracy varies across PCA dimensions and algorithms, highlighting the best-performing combinations:

```
pivot = df_hyper.pivot_table(
    index="PCA_dim",
    columns="algorithm",
    values="Val_Accuracy",
    aggfunc="max"
)

plt.figure(figsize=(10,6))
sns.heatmap(pivot, annot=True, cmap="viridis")
plt.title("Best Validation Accuracy by PCA Dimension and Algorithm")
plt.show()
```

Moshiur Howlader



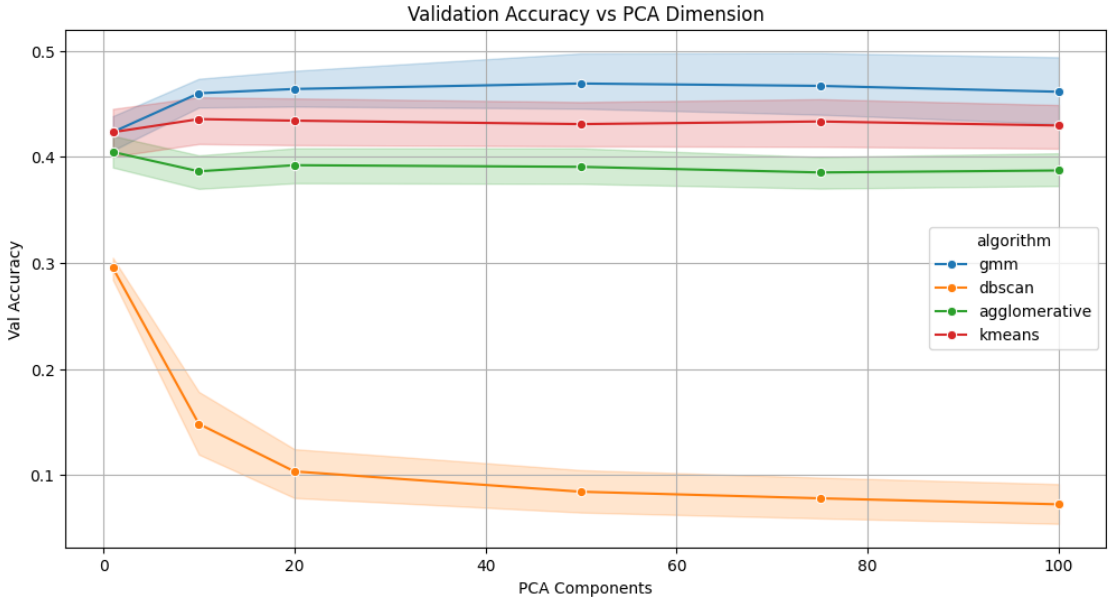
The table below shows the best-performing PCA dimension for each algorithm based on the highest validation accuracy:

```
best_pca_per_algo = (
    df_hyper.sort_values("Val_Accuracy", ascending=False)
            .groupby("algorithm")
            .head(1)
)

display(best_pca_per_algo)
```

```
   features      algorithm  PCA_dim                 params  Train_ARI  \
0       rgb            gmm      100          k=5, cov=full   0.266258
36    canny         dbscan       10  eps=3.0, min_samples=5   0.041806
38    canny  agglomerative       50       k=5, linkage=ward   0.062544
52    sobel         kmeans       10                    k=5   0.067218

    Train_NMI  Train_Accuracy  Val_Accuracy
0    0.300976        0.739234      0.750000
36   0.041640        0.220574      0.503333
38   0.071056        0.510731      0.500667
52   0.078765        0.493985      0.490667
```

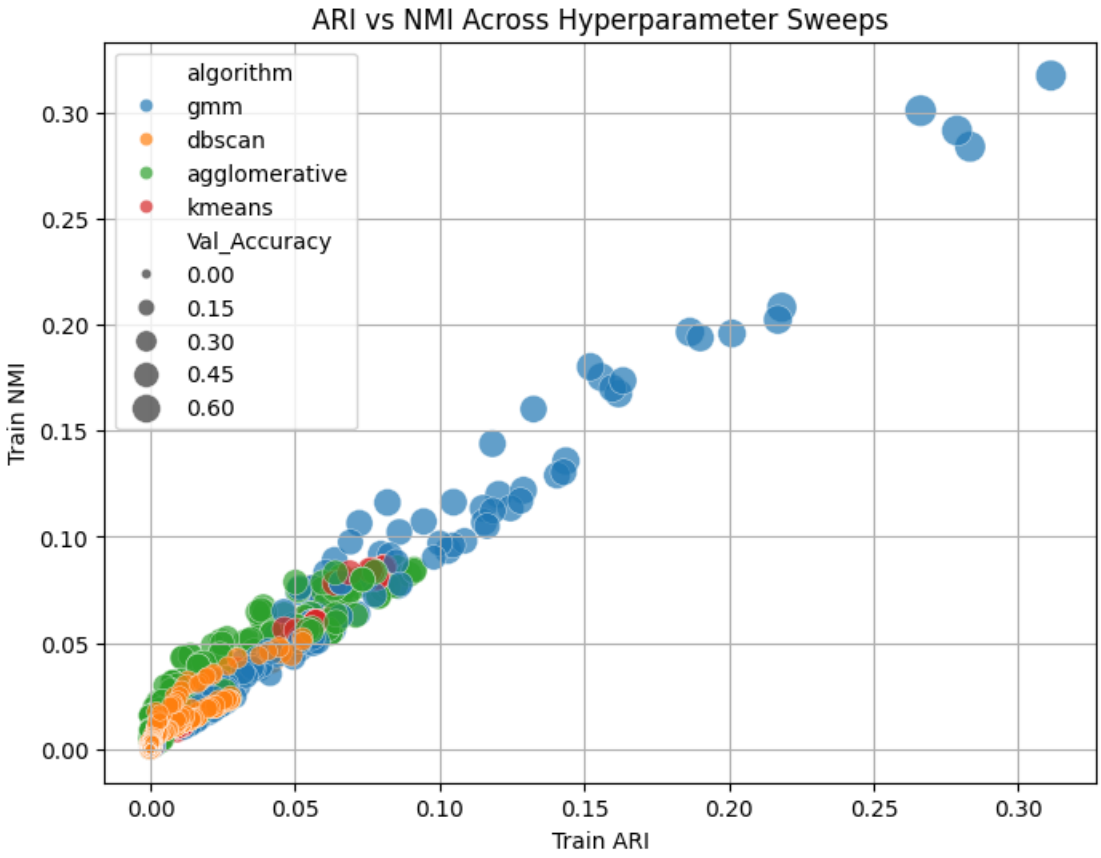
The line plot below illustrates how validation accuracy changes with PCA dimensionality for each algorithm, revealing performance trends across component counts:

```
plt.figure(figsize=(12,6))
sns.lineplot(data=df_hyper, x="PCA_dim", y="Val_Accuracy", hue="algorithm", m
arker="o")

plt.title("Validation Accuracy vs PCA Dimension")
plt.ylabel("Val Accuracy")
plt.xlabel("PCA Components")
plt.grid(True)
plt.show()
```



The scatter plot below visualizes the relationship between ARI and NMI across hyperparameter sweeps, with point size indicating validation accuracy for each algorithm:

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

sns.scatterplot(
    data=df_hyper,
    x="Train_ARI",
    y="Train_NMI",
    hue="algorithm",
    size="Val_Accuracy",
    sizes=(20,200),
    alpha=0.7
)

plt.title("ARI vs NMI Across Hyperparameter Sweeps")
plt.xlabel("Train ARI")
plt.ylabel("Train NMI")
plt.grid(True)
plt.show()
```



4.1 Conclusion — Classical Unsupervised Learning

The unsupervised experiments revealed clear performance differences across feature types, clustering algorithms, and PCA dimensionalities. Contrary to the initial expectation that edge-based features might simplify clustering by reducing color variation, **RGB features consistently produced the strongest results**. In particular, **Gaussian Mixture Models (GMM)** using **75–100 PCA components** achieved the highest performance, reaching **0.72–0.75 validation accuracy**, significantly outperforming all edge-based representations.

Edge filters such as Sobel, Canny, and Laplacian produced moderate results, typically in the **0.50–0.60** accuracy range depending on the model. While these representations highlight structural boundaries and reduce color noise, they remove too much semantic information—textures, patterns, and color gradients—making it difficult for clustering algorithms to reliably separate cat, dog, and wild classes.

Among the algorithms tested:

- **GMM was the strongest performer**, benefiting from flexible covariance modeling that can capture elongated and elliptical cluster shapes.
- **Agglomerative Clustering** and **K-Means were moderate performers**, both reaching validation accuracies around **0.50** in their best configurations. Their

limitations stem from simplified distance assumptions—spherical clusters for K-Means and irreversible merge decisions for Agglomerative—making them less effective for complex image embeddings.

- **DBSCAN was the weakest performer overall.** It showed one unusually high result at **PCA = 10** (accuracy 0.50), but fell to **0.22–0.34** for all other PCA dimensions. This instability reflects how density-based clustering breaks down as dimensionality changes, making DBSCAN unsuitable for these image features.

Overall, while the top-performing configuration (RGB + GMM) delivered a surprisingly strong **~75%** accuracy for an unsupervised pipeline, classical clustering methods remain fundamentally limited. Flattened pixel vectors, simple edge descriptors, and linear PCA projections cannot capture the rich, nonlinear visual patterns present in natural images. These findings underscore the need for more expressive feature extractors—particularly **Convolutional Neural Networks (CNNs)**—which can learn hierarchical spatial representations far beyond the capability of traditional unsupervised approaches.

4.2 Part B – CNN Training

4.2.1 - Primer on CNN

A brief primer on CNNs is included here to support the analysis (a more in-depth treatment appears in the next ML course).

Introduction to Convolutional Neural Networks (CNNs)

To complement the unsupervised approaches, a supervised deep learning model called Convolutional Neural Networks (CNN) is trained to establish a performance benchmark for image classification. CNNs exploit spatial locality in images by sharing filters across the input, which significantly reduces parameter count compared to fully connected networks and allows the model to efficiently learn local visual patterns.

CNNs are deep learning architectures designed specifically for image data. Their core operation, the **convolution**, uses small learnable filters that slide across the image and compute weighted sums over local pixel neighborhoods. This highlights spatial structures such as edges, corners, and textures. Early layers usually learn simple patterns (e.g., vertical or horizontal edges), while deeper layers gradually identify more complex structures such as textures, contours, and object parts.

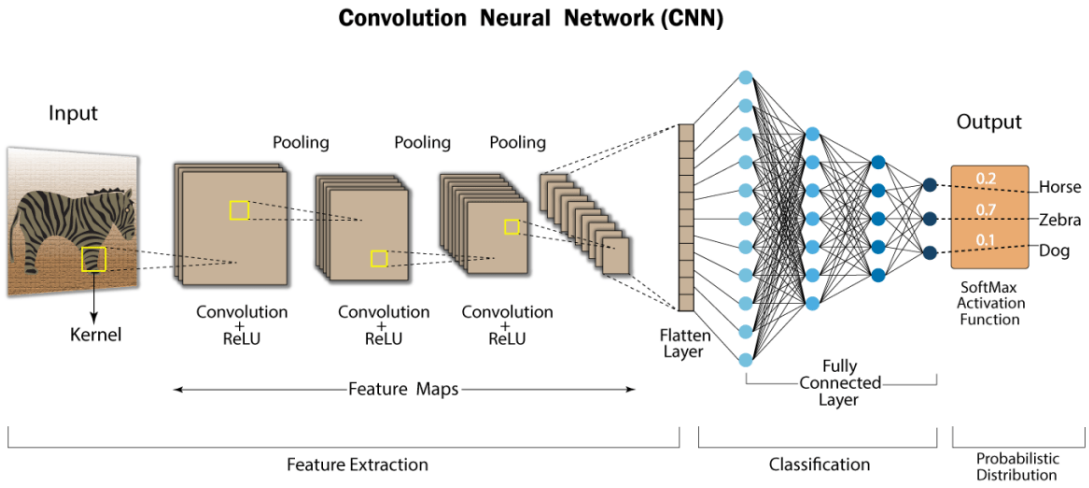
As the image moves through each CNN layer, the network progressively learns more detailed and abstract information. Early layers detect basic edges and lines; middle layers combine these into larger shapes and textures; and deeper layers learn high-level features that describe the object itself. After this feature-extraction process, the final feature maps

are flattened and passed into fully connected layers that perform classification. The final Softmax layer converts these outputs into class probabilities.

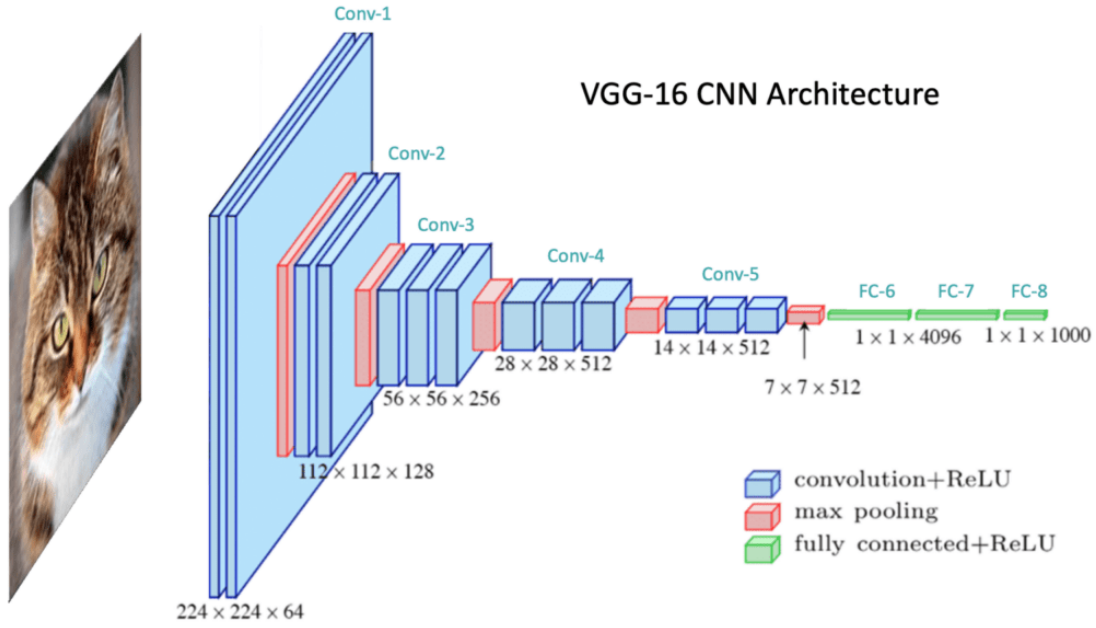
In this project, the CNN leverages this hierarchical pipeline to automatically learn meaningful features from both RGB images and edge-based representations. This enables accurate classification into *cat*, *dog*, and *wild* categories, outperforming the unsupervised clustering methods in both accuracy and consistency.

CNN Architecture Diagrams

The diagram below illustrates the general CNN architecture:



A second diagram is provided to further visualize the feature-extraction pipeline:



Further Readings

- Wikipedia - Convolutional Neural Network
- Learn OpenCV - Understanding Convolutional Neural Networks
- Medium Article on Convolutional Neural Networks

4.2.2 - Training CNN Models

In this section, a CNN is trained to both improve overall classification performance and assess how well the model handles images processed with various edge filters.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt

# ====================================================
# 1. LABEL ENCODING
# ====================================================

le = LabelEncoder()
y_train = le.fit_transform(labels_train)
y_val   = le.transform(labels_val)

print("Classes:", le.classes_)

# ====================================================
# 2. PREPARE ALL 4 INPUT SETS
# ====================================================

datasets = {
    "rgb": data_small.astype("float32") / 255.0,
    "canny": data_edges_canny_small.astype("float32") / 255.0,
    "sobel": data_edges_sobel_small.astype("float32") / 255.0,
    "laplacian": data_edges_laplacian_small.astype("float32") / 255.0
}

datasets_val = {
    "rgb": data_small_val.astype("float32") / 255.0,
    "canny": data_edges_canny_small_val.astype("float32") / 255.0,
    "sobel": data_edges_sobel_small_val.astype("float32") / 255.0,
    "laplacian": data_edges_laplacian_small_val.astype("float32") / 255.0
}
```

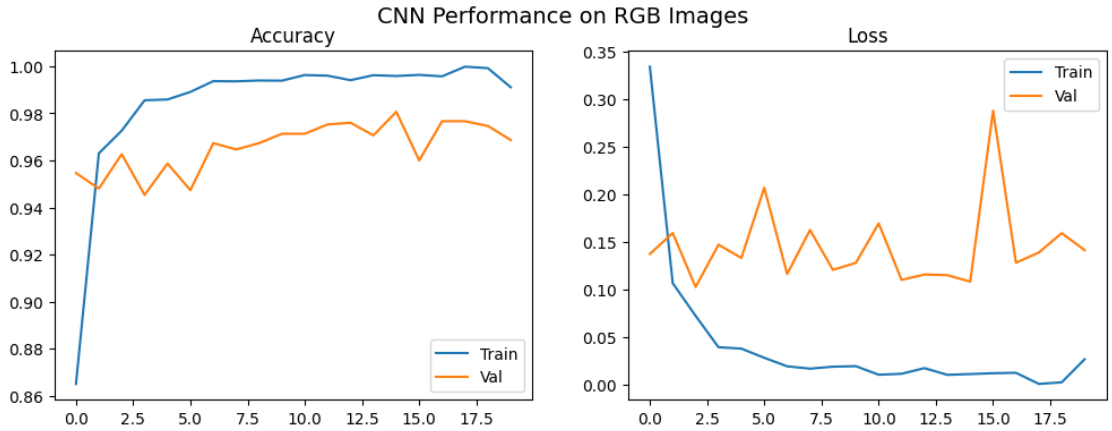
```
# Expand grayscale to 3 channels so model input stays the same (channel repli
cation, all three channels has the same image)
for key in ["canny", "sobel", "laplacian"]:
    datasets[key] = np.stack([datasets[key]] * 3, axis=-1)
    datasets_val[key] = np.stack([datasets_val[key]] * 3, axis=-1)

# Final shapes should be (N,128,128,3)
for k, v in datasets.items():
    print(k, v.shape)



# ==================================================
# 3. DEFINE A FUNCTION TO BUILD THE CNN
# ==================================================

# Using three convolutional blocks
def create_cnn():
    model = models.Sequential([
        layers.Conv2D(32, 3, activation='relu', padding='same', input_shape=
(128,128,3)),
        layers.MaxPooling2D(),

        layers.Conv2D(64, 3, activation='relu', padding='same'),
        layers.MaxPooling2D(),

        layers.Conv2D(128, 3, activation='relu', padding='same'),
        layers.MaxPooling2D(),

        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dense(3, activation='softmax')
    ])

    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )
    return model



# ==================================================
# 4. TRAIN CNN FOR EACH FEATURE TYPE
# ==================================================

results = {}

for feature_name in datasets.keys():
```

```
    print("\n====================================")
    print(f"Training CNN on: {feature_name.upper()}")
    print("====================================")

    X_train = datasets[feature_name]
    X_val   = datasets_val[feature_name]

    model = create_cnn()

    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=20,
        batch_size=32,
        verbose=1
    )

    # Store metrics
    results[feature_name] = history.history

    # Plot curves
    plt.figure(figsize=(12,4))
    plt.suptitle(f"CNN Performance on {feature_name.upper()} Images", fontsiz
e=14)

    # Accuracy
    plt.subplot(1,2,1)
    plt.plot(history.history['accuracy'], label='Train')
    plt.plot(history.history['val_accuracy'], label='Val')
    plt.title("Accuracy")
    plt.legend()

    # Loss
    plt.subplot(1,2,2)
    plt.plot(history.history['loss'], label='Train')
    plt.plot(history.history['val_loss'], label='Val')
    plt.title("Loss")
    plt.legend()

    plt.show()
```

```
Classes: ['cat' 'dog' 'wild']
rgb (14630, 128, 128, 3)
canny (14630, 128, 128, 3)
sobel (14630, 128, 128, 3)
laplacian (14630, 128, 128, 3)

====================================
```

```
Training CNN on: RGB
====================================

c:\Users\howla\AppData\Local\Programs\Python\Python312\Lib\site-packages\kera
s\src\layers\convolutional\base_conv.py:113: UserWarning: Do not pass an `inp
ut_shape`/`input_dim` argument to a layer. When using Sequential models, pref
er using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 26s 56ms/step - accuracy: 0.8649 - loss: 0.3341
- val_accuracy: 0.9547 - val_loss: 0.1374
Epoch 2/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 55ms/step - accuracy: 0.9630 - loss: 0.1067
- val_accuracy: 0.9480 - val_loss: 0.1594
Epoch 3/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9727 - loss: 0.0724
- val_accuracy: 0.9627 - val_loss: 0.1027
Epoch 4/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 52ms/step - accuracy: 0.9856 - loss: 0.0394
- val_accuracy: 0.9453 - val_loss: 0.1471
Epoch 5/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 52ms/step - accuracy: 0.9859 - loss: 0.0380
- val_accuracy: 0.9587 - val_loss: 0.1332
Epoch 6/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9891 - loss: 0.0283
- val_accuracy: 0.9473 - val_loss: 0.2070
Epoch 7/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9937 - loss: 0.0194
- val_accuracy: 0.9673 - val_loss: 0.1163
Epoch 8/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9936 - loss: 0.0169
- val_accuracy: 0.9647 - val_loss: 0.1626
Epoch 9/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9940 - loss: 0.0190
- val_accuracy: 0.9673 - val_loss: 0.1207
Epoch 10/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9939 - loss: 0.0195
- val_accuracy: 0.9713 - val_loss: 0.1280
Epoch 11/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9963 - loss: 0.0106
- val_accuracy: 0.9713 - val_loss: 0.1695
Epoch 12/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9960 - loss: 0.0116
- val_accuracy: 0.9753 - val_loss: 0.1102
Epoch 13/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9941 - loss: 0.0175
- val_accuracy: 0.9760 - val_loss: 0.1158
Epoch 14/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9962 - loss: 0.0104
```

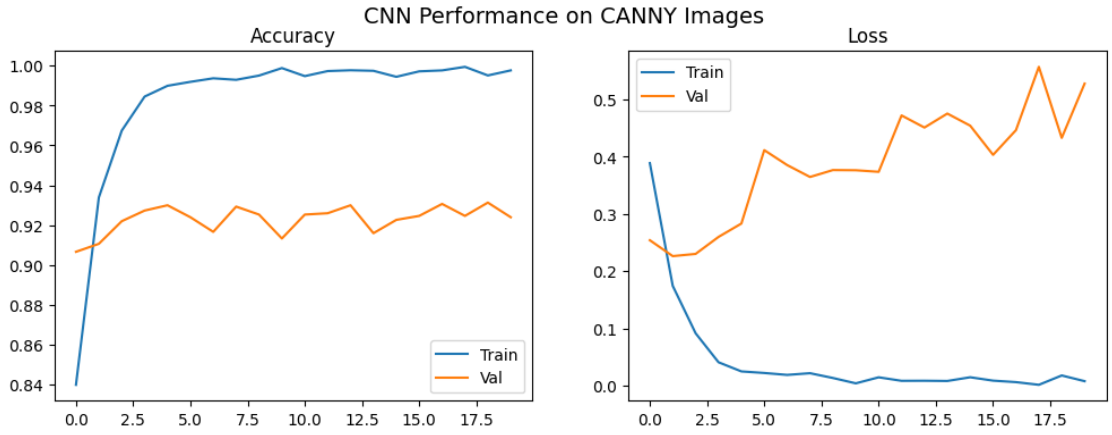
```
- val_accuracy: 0.9707 - val_loss: 0.1151
Epoch 15/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 55ms/step - accuracy: 0.9959 - loss: 0.0113
- val_accuracy: 0.9807 - val_loss: 0.1083
Epoch 16/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 55ms/step - accuracy: 0.9964 - loss: 0.0121
- val_accuracy: 0.9600 - val_loss: 0.2879
Epoch 17/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 56ms/step - accuracy: 0.9958 - loss: 0.0126
- val_accuracy: 0.9767 - val_loss: 0.1282
Epoch 18/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 26s 57ms/step - accuracy: 0.9999 - loss: 8.8945e
-04 - val_accuracy: 0.9767 - val_loss: 0.1389
Epoch 19/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 26s 57ms/step - accuracy: 0.9992 - loss: 0.0026
- val_accuracy: 0.9747 - val_loss: 0.1593
Epoch 20/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 55ms/step - accuracy: 0.9911 - loss: 0.0268
- val_accuracy: 0.9687 - val_loss: 0.1414
```




```
====================================
Training CNN on: CANNY
====================================
Epoch 1/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 26s 55ms/step - accuracy: 0.8400 - loss: 0.3885
- val_accuracy: 0.9067 - val_loss: 0.2540
Epoch 2/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9339 - loss: 0.1744
- val_accuracy: 0.9107 - val_loss: 0.2263
Epoch 3/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9674 - loss: 0.0917
- val_accuracy: 0.9220 - val_loss: 0.2301
Epoch 4/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9845 - loss: 0.0411
- val_accuracy: 0.9273 - val_loss: 0.2598
```

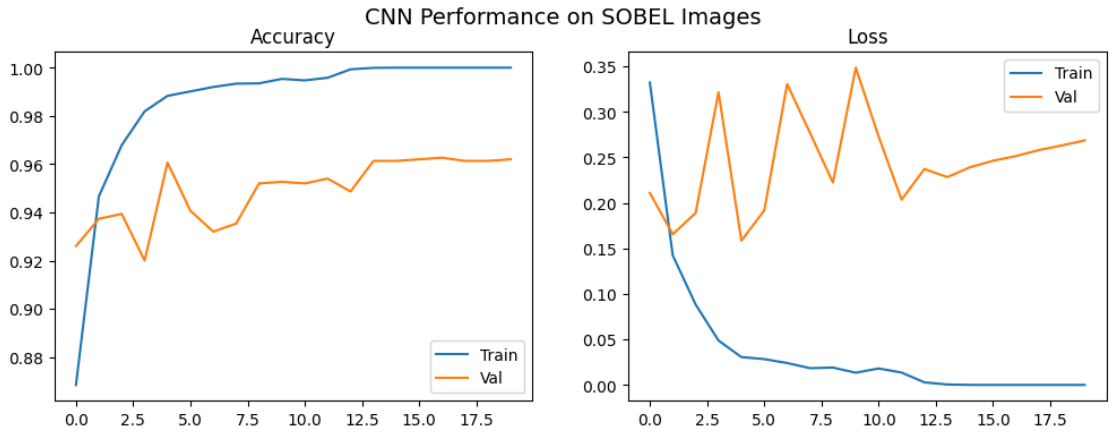
```
Epoch 5/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9899 - loss: 0.0251
- val_accuracy: 0.9300 - val_loss: 0.2831
Epoch 6/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9919 - loss: 0.0224
- val_accuracy: 0.9240 - val_loss: 0.4112
Epoch 7/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9936 - loss: 0.0190
- val_accuracy: 0.9167 - val_loss: 0.3852
Epoch 8/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 52ms/step - accuracy: 0.9929 - loss: 0.0220
- val_accuracy: 0.9293 - val_loss: 0.3644
Epoch 9/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9950 - loss: 0.0136
- val_accuracy: 0.9253 - val_loss: 0.3764
Epoch 10/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9988 - loss: 0.0043
- val_accuracy: 0.9133 - val_loss: 0.3761
Epoch 11/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9947 - loss: 0.0148
- val_accuracy: 0.9253 - val_loss: 0.3734
Epoch 12/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9973 - loss: 0.0087
- val_accuracy: 0.9260 - val_loss: 0.4718
Epoch 13/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9977 - loss: 0.0088
- val_accuracy: 0.9300 - val_loss: 0.4506
Epoch 14/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9974 - loss: 0.0084
- val_accuracy: 0.9160 - val_loss: 0.4750
Epoch 15/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9944 - loss: 0.0149
- val_accuracy: 0.9227 - val_loss: 0.4539
Epoch 16/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9971 - loss: 0.0090
- val_accuracy: 0.9247 - val_loss: 0.4030
Epoch 17/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9976 - loss: 0.0064
- val_accuracy: 0.9307 - val_loss: 0.4461
Epoch 18/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9994 - loss: 0.0017
- val_accuracy: 0.9247 - val_loss: 0.5566
Epoch 19/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9951 - loss: 0.0180
- val_accuracy: 0.9313 - val_loss: 0.4326
Epoch 20/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 53ms/step - accuracy: 0.9976 - loss: 0.0081
- val_accuracy: 0.9240 - val_loss: 0.5273
```

Moshiur Howlader




```
==================================
Training CNN on: SOBEL
==================================
Epoch 1/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 26s 55ms/step - accuracy: 0.8684 - loss: 0.3322
- val_accuracy: 0.9260 - val_loss: 0.2109
Epoch 2/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9466 - loss: 0.1424
- val_accuracy: 0.9373 - val_loss: 0.1656
Epoch 3/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9679 - loss: 0.0884
- val_accuracy: 0.9393 - val_loss: 0.1889
Epoch 4/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9819 - loss: 0.0488
- val_accuracy: 0.9200 - val_loss: 0.3216
Epoch 5/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9883 - loss: 0.0306
- val_accuracy: 0.9607 - val_loss: 0.1585
Epoch 6/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 53ms/step - accuracy: 0.9902 - loss: 0.0284
- val_accuracy: 0.9407 - val_loss: 0.1918
Epoch 7/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9920 - loss: 0.0241
- val_accuracy: 0.9320 - val_loss: 0.3307
Epoch 8/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9934 - loss: 0.0184
- val_accuracy: 0.9353 - val_loss: 0.2769
Epoch 9/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9935 - loss: 0.0191
- val_accuracy: 0.9520 - val_loss: 0.2225
Epoch 10/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9954 - loss: 0.0134
- val_accuracy: 0.9527 - val_loss: 0.3486
Epoch 11/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9947 - loss: 0.0181
```

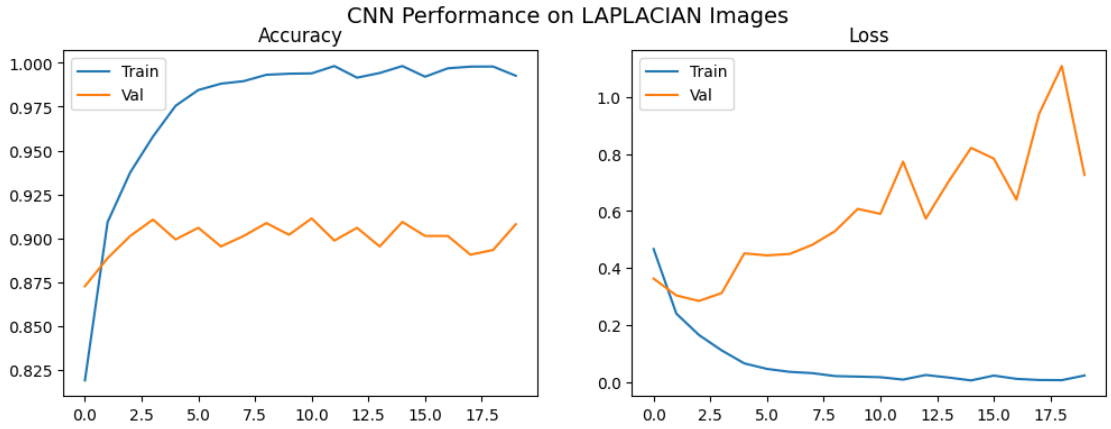
```
- val_accuracy: 0.9520 - val_loss: 0.2727
Epoch 12/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9958 - loss: 0.0136
- val_accuracy: 0.9540 - val_loss: 0.2036
Epoch 13/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9993 - loss: 0.0028
- val_accuracy: 0.9487 - val_loss: 0.2373
Epoch 14/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9999 - loss: 4.7895e
-04 - val_accuracy: 0.9613 - val_loss: 0.2284
Epoch 15/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 1.0000 - loss: 3.7665e
-05 - val_accuracy: 0.9613 - val_loss: 0.2393
Epoch 16/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 1.0000 - loss: 1.9109e
-05 - val_accuracy: 0.9620 - val_loss: 0.2463
Epoch 17/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 1.0000 - loss: 1.3175e
-05 - val_accuracy: 0.9627 - val_loss: 0.2515
Epoch 18/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 1.0000 - loss: 9.7284e
-06 - val_accuracy: 0.9613 - val_loss: 0.2581
Epoch 19/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 1.0000 - loss: 7.3307e
-06 - val_accuracy: 0.9613 - val_loss: 0.2631
Epoch 20/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 1.0000 - loss: 5.6170e
-06 - val_accuracy: 0.9620 - val_loss: 0.2686
```




```
==================================
Training CNN on: LAPLACIAN
==================================
Epoch 1/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 26s 55ms/step - accuracy: 0.8190 - loss: 0.4667
- val_accuracy: 0.8727 - val_loss: 0.3630
```

```
Epoch 2/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9092 - loss: 0.2403
- val_accuracy: 0.8887 - val_loss: 0.3040
Epoch 3/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9376 - loss: 0.1650
- val_accuracy: 0.9013 - val_loss: 0.2847
Epoch 4/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9580 - loss: 0.1111
- val_accuracy: 0.9107 - val_loss: 0.3124
Epoch 5/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9755 - loss: 0.0655
- val_accuracy: 0.8993 - val_loss: 0.4517
Epoch 6/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9845 - loss: 0.0461
- val_accuracy: 0.9060 - val_loss: 0.4444
Epoch 7/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9881 - loss: 0.0358
- val_accuracy: 0.8953 - val_loss: 0.4495
Epoch 8/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9895 - loss: 0.0313
- val_accuracy: 0.9013 - val_loss: 0.4818
Epoch 9/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9932 - loss: 0.0209
- val_accuracy: 0.9087 - val_loss: 0.5296
Epoch 10/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9938 - loss: 0.0191
- val_accuracy: 0.9020 - val_loss: 0.6076
Epoch 11/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9940 - loss: 0.0171
- val_accuracy: 0.9113 - val_loss: 0.5904
Epoch 12/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9982 - loss: 0.0087
- val_accuracy: 0.8987 - val_loss: 0.7733
Epoch 13/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9915 - loss: 0.0248
- val_accuracy: 0.9060 - val_loss: 0.5737
Epoch 14/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 54ms/step - accuracy: 0.9942 - loss: 0.0160
- val_accuracy: 0.8953 - val_loss: 0.7027
Epoch 15/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 53ms/step - accuracy: 0.9982 - loss: 0.0058
- val_accuracy: 0.9093 - val_loss: 0.8218
Epoch 16/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9921 - loss: 0.0227
- val_accuracy: 0.9013 - val_loss: 0.7841
Epoch 17/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 25s 53ms/step - accuracy: 0.9969 - loss: 0.0113
- val_accuracy: 0.9013 - val_loss: 0.6400
Epoch 18/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9978 - loss: 0.0072
```

```
- val_accuracy: 0.8907 - val_loss: 0.9416
Epoch 19/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9979 - loss: 0.0067
- val_accuracy: 0.8933 - val_loss: 1.1090
Epoch 20/20
458/458 ━━━━━━━━━━━━━━━━━━━━ 24s 53ms/step - accuracy: 0.9926 - loss: 0.0230
- val_accuracy: 0.9080 - val_loss: 0.7268
```



Why CNN Hyperparameter Tuning Is Not Included

Running a full hyperparameter search for a CNN is unnecessary for this project. CNN training is computationally demanding and typically relies on NVIDIA GPUs with CUDA acceleration. Since this workstation does not have an NVIDIA GPU, running dozens of different CNN configurations would take an impractical amount of time.

More importantly, the baseline CNN already achieves strong validation accuracy (≈97–98%), which clearly demonstrates the intended point: supervised CNNs outperform all classical unsupervised clustering methods on image classification tasks.

Further tuning would not meaningfully change this conclusion. The goal of this experiment is to compare learning paradigms, not to optimize a deep learning architecture. The current results already provide all the evidence needed, while additional tuning would only increase runtime and complexity without adding new insight.

4.2.3 - Summary of the output for CNN:

Note: CNN training is non-deterministic, so results may vary slightly between runs.

Input Type	Train Accuracy	Validation Accuracy	Difficulty	Notes
RGB	~0.99	**0.97–0.98**	Easiest	Best texture, color, and

Input Type	Train Accuracy	Validation Accuracy	Difficulty	Notes
				shape information
Sobel	~1.00	**0.95–0.96**	Medium	Strong gradients preserve useful structure
Laplacian	~1.00	**0.89–0.91**	Harder	High-frequency noise, reduced detail
Canny	~1.00	**0.92–0.93**	Hardest	Very sparse edges, minimal texture

4.2 Conclusion and Key Insights

The CNN results reveal a **clear, consistent performance hierarchy**:

RGB -> Sobel -> Canny -> Laplacian

This ranking aligns directly with the type and amount of visual information each representation provides to the CNN.

Why CNNs Perform Best on Richer Representations

CNNs learn features hierarchically:

1. **Early layers** detect edges, corners, and color gradients.

2. **Middle layers** assemble these into textures and shapes.

3. **Deep layers** extract semantic object-level cues.

Because of this pipeline, CNNs benefit most from inputs containing **dense, continuous visual information**:

- **RGB** supplies rich color variation, smooth gradients, shading, and texture—ideal for convolutional filters.

- **Sobel** preserves strong edges and directional gradients, enabling the CNN to still build solid mid-level features.

- **Canny** provides only sparse contour information, giving the network fewer pixels to learn from and weakening its ability to form high-level abstractions.

- **Laplacian** exaggerates high-frequency components, creating noisy edges that distort the shape and texture cues CNNs rely on.

Why Train Accuracy Is ~1.00 Even for Weak Inputs

CNNs are powerful enough to memorize the training images, even when the inputs are weak. Because of this, the training accuracy can reach nearly 100% without actually meaning the model learned well. This is why **validation accuracy** is the real measure of performance.

Final Conclusion

CNNs rely heavily on the richness and continuity of the input signal.
The more detailed and informative the representation, the more reliably the model can form meaningful hierarchical features. As inputs become sparser or noisier, CNNs lose the ability to extract mid- and high-level patterns, causing a steady drop in classification performance.

---

5. Results, Analysis, and Conclusions

This section summarizes the performance of all models used in the project and integrates the findings from both the unsupervised clustering methods and the supervised CNN. The goal is to compare how well each approach separates the three classes (cat, dog, wild), evaluate the impact of different image feature representations (RGB and edge-based inputs), and determine whether the initial hypotheses formed during the EDA phase hold

true (the hypothesis that clustering will be influenced more by **texture**, **shape**, and **edge patterns** rather than by color alone). The subsections below present the results, analyze key patterns, and conclude with the broader insights gained from the project.

5.1 Unsupervised Results Summary

In Section 4.1, classical unsupervised algorithms such as K-Means, Gaussian Mixture Models (GMM), DBSCAN, and Agglomerative (Hierarchical) Clustering were applied to the **AFHQ** animal-faces dataset. All images (RGB, Canny, Sobel, Laplacian) were resized to `128×128`, flattened, normalized to the `[0–1]` range, and reduced in dimensionality using PCA before being passed into the clustering models. Training accuracy was computed by assigning each cluster to its most frequent label, and validation accuracy was measured using unseen data to assess how well the learned clusters generalized.

Across all experiments, Gaussian Mixture Models (GMM) with RGB features produced the strongest results, achieving validation accuracies in the 0.72–0.75 range. This was substantially higher than the edge-based feature sets and confirms that richer texture and color information gave GMM a stronger foundation for separation. The best configurations typically used 75–100 PCA components, suggesting that preserving mid-to-high dimensional variance was important for capturing semantic differences in the images.

The other clustering algorithms showed moderate performance:

- Agglomerative Clustering reached at most ~0.50 validation accuracy, with the best configuration using Canny edges + PCA(50).
- K-Means also performed moderately across the board, with its best configuration (Sobel + PCA(10)) achieving ~0.49 validation accuracy, close to random guessing for a three-class task.
- DBSCAN performed the worst overall, reaching ~0.50 validation accuracy only in the Canny + PCA(10) configuration. In all other cases, DBSCAN dropped to 0.20–0.30, especially in higher-dimensional representations, where it often collapsed into a single cluster or labeled large portions of the data as noise.

Overall, even with broad hyperparameter sweeps, most unsupervised methods plateaued around 0.48–0.55 accuracy, with GMM + RGB being the only strong outlier at ~0.75. These results demonstrate both the potential and the limitations of classical clustering: while GMM can exploit richer RGB structure effectively, linear PCA projections and simple edge-based descriptors are insufficient to fully capture the semantic complexity needed for reliable cat/dog/wild separation.

5.2 CNN Results Summary

The CNN provided a strong supervised baseline and consistently outperformed all classical unsupervised methods across every input type. Using three convolutional blocks followed by fully connected layers, the model achieved rapid convergence and stable

learning on both RGB images and edge-based representations of the **AFHQ** dataset. As expected, performance strongly correlated with the amount and quality of visual information available in each feature set.

RGB images produced the highest accuracy, reaching approximately **0.97–0.98** validation accuracy. These results reflect the CNN's ability to leverage full color, texture, shading, and spatial structure when learning hierarchical features. Edge-based inputs yielded progressively lower accuracy depending on how much information they removed: **Sobel (~0.95–0.96)** retained strong gradients and performed well; **Canny (~0.92–0.93)** was more sparse but still usable; **Laplacian (~0.89–0.91)** introduced high-frequency noise and led to the weakest performance.

Training accuracy reached ~1.00 across all feature types, which is expected for CNNs of this capacity on moderate-sized datasets. However, validation accuracy clearly differentiated the usefulness of each representation. The results confirm that CNNs rely heavily on rich, continuous visual patterns to construct meaningful multi-level features, and degrade gracefully as the input becomes sparser or noisier.

Overall, the CNN established a high, reliable performance ceiling and illustrated the significant advantage of deep learning over classical clustering for this image classification task.

5.3 Comparison: Unsupervised vs CNN

The performance gap between classical unsupervised clustering and the supervised CNN is summarized in the table below. While GMM with RGB features reached a moderately strong 0.75 validation accuracy, all other clustering configurations performed substantially lower. In contrast, the CNN achieved significantly higher accuracy across all input types, with RGB performance reaching 0.97–0.98.

This table provides a direct comparison of the strongest results from both approaches:

Method Type	Feature Input	Best Algorithm	Best PCA Dim	Validation Accuracy	Notes
Unsupervised	RGB	GMM	100	**0.75**	Best classical method
Unsupervised	Sobel	K-Means	10	0.49	Lacks color/texture

Method Type	Feature Input	Best Algorithm	Best PCA Dim	Validation Accuracy	Notes
Unsupervised	Canny	Agglomerative	50	0.50	Sparse edges
Unsupervised	Canny	DBSCAN	10	0.50	Density-based, limited
CNN (Supervised)	RGB	CNN	—	**0.97–0.98**	Best overall
CNN	Sobel	CNN	—	0.95–0.96	Strong gradients
CNN	Canny	CNN	—	0.92–0.93	Sparse edges
CNN	Laplacian	CNN	—	0.89–0.91	Noisy gradients

The contrast between the classical unsupervised methods and the supervised CNN highlights the fundamental difference in their representational power. Even under their best configurations, the unsupervised models achieved only partial class separation, with **GMM + RGB** reaching a peak validation accuracy of **≈0.75**. All other combinations—including K-Means, Agglomerative, DBSCAN, and edge-based features—performed notably worse, typically falling in the **0.48–0.50** range. These algorithms rely heavily on linear projections (PCA), distance metrics, and simple statistical assumptions, none of which are sufficient to capture the nonlinear, high-level visual structure present in the **AFHQ** dataset.

In contrast, the CNN demonstrated a far more expressive and robust feature-learning capability. By learning hierarchical representations directly from pixel data—edges, textures, shapes, and high-level semantic patterns—the CNN achieved **substantially higher accuracy across all input types**, with RGB validation accuracy reaching **0.97–0.98**. Even edge-filtered inputs like Sobel, Canny, and Laplacian, which lose significant

structural detail, still outperformed every unsupervised method once processed through the CNN's convolutional pipeline.

Overall, the supervised CNN not only achieved higher accuracy but also exhibited significantly better generalization and stability. The results clearly show that **deep learning captures complex spatial relationships that classical clustering cannot**, establishing CNNs as the superior approach for image classification in this dataset.

5.4 Edge-based Hypothesis: Did the Data Match Our Expectations?

The initial hypothesis proposed that edge-based representations (Sobel, Canny, Laplacian) would preserve the essential structural information—such as contours, gradients, and texture transitions—while reducing the impact of background color variation and lighting. It was expected that these filtered inputs would perform similarly to RGB or only slightly worse due to a modest loss of information.

The results partially supported this reasoning. Edge filters did retain meaningful structure and allowed both clustering algorithms and the CNN to form reasonable group distinctions. However, the performance gap between edge-based inputs and RGB was larger than anticipated. For unsupervised learning, RGB + GMM achieved a validation accuracy of ≈0.75, significantly higher than any edge-based configuration, which generally fell in the 0.50–0.60 range. A similar pattern emerged in the CNN experiments: while Sobel, Canny, and Laplacian inputs produced respectable accuracy, they consistently trailed the RGB baseline, following a clear ordering aligned with the richness of visual information preserved:

RGB -> Sobel -> Canny -> Laplacian

These outcomes suggest that while edge filtering does reduce irrelevant variation, the loss of texture, fine gradients, and subtle color cues ultimately harms performance more than expected. Thus, the hypothesis was only weakly supported: edge-based inputs remained usable but did not match the effectiveness of RGB in either unsupervised or supervised settings.

5.5 Final Conclusions

The results across all experiments show a clear distinction between classical unsupervised methods and supervised deep learning. Traditional clustering approaches achieved limited class separability, with only **GMM + RGB** producing moderately strong results at **≈0.75** accuracy, and all other configurations lagging significantly behind. These findings highlight the limitations of relying on linear projections, simple edge filters, and distance-based similarity metrics for complex natural images.

In contrast, the CNN established a high and stable performance ceiling, reaching **0.97–0.98** accuracy on RGB inputs and demonstrating strong generalization even on edge-based images. The network's ability to learn hierarchical, nonlinear spatial features enabled it to capture the semantic distinctions that classical clustering could not. Overall, the study

reinforces that **deep learning provides a substantially more expressive and reliable foundation for image classification**, especially when working with diverse and visually complex datasets like **AFHQ**.

6. Future Improvements and Areas to Explore

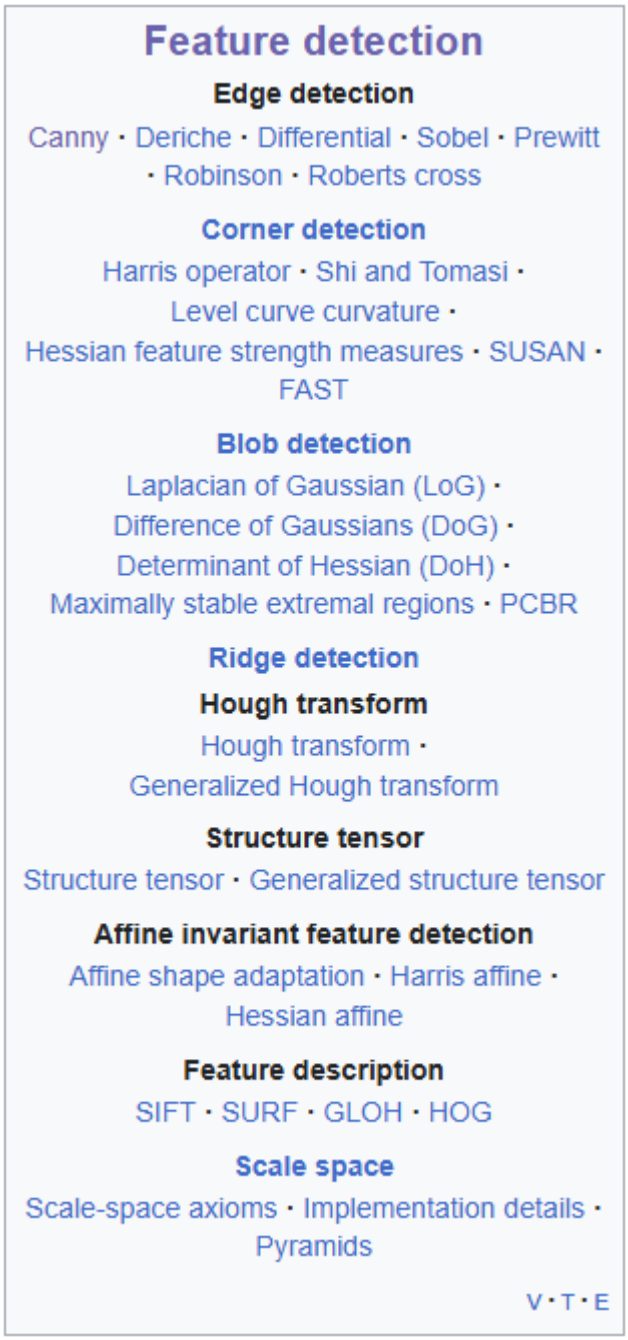
Several extensions and improvements could be explored to build on the findings of this project:

- **Dimensionality Reduction Beyond PCA:**
  Nonlinear methods such as **UMAP** or **t-SNE** could be evaluated for their ability to produce more meaningful low-dimensional spaces prior to clustering.

- **Hyperparameter Optimization for CNNs:**
  While tuning was intentionally out of scope, future work could evaluate alternative batch sizes, learning rate schedules, optimizers, or regularization techniques (e.g., dropout, weight decay) to improve performance and training stability.

- **Alternative Edge or Texture Representations:**
  Richer hand-crafted descriptors—such as **HOG**, **SIFT**, or **Gabor filters**—may offer stronger unsupervised performance than Sobel, Canny, or Laplacian. Exploring the broader collections of classical computer vision techniques found in research surveys and Wikipedia catalogues may also inspire additional feature-engineering

ideas:

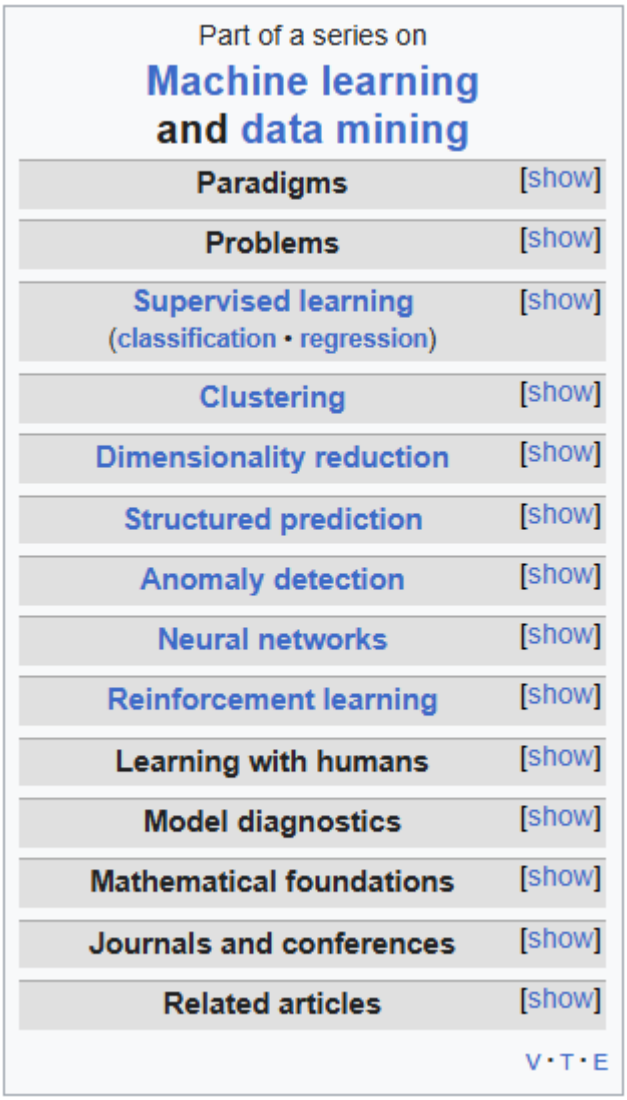


- **Advanced Digital Image Processing Techniques:**
  Additional image enhancement, denoising, segmentation, and feature-extraction methods could be explored to better understand trade-offs between computational cost and model accuracy—particularly in real-time applications where efficiency is critical.

- **Advanced CNN Architectures and Techniques:**
  More expressive deep-learning architectures such as **ResNet50V2**, **ResNet152V2**,

Xception, **InceptionV3**, and **MobileNetV2** may significantly improve classification accuracy and generalization.
(See: Comparative Study of CNN Architectures and Advancements in Deep Learning Architectures)

- **Broader Machine Learning and Computer Vision Topics:**
  Numerous areas remain open for future exploration, including generative models, self-supervised learning, object detection, segmentation, pose estimation, and more.



7. References and Acknowledgments

1. **Animal Faces-HQ (AFHQ) Dataset (Kaggle):**
   https://www.kaggle.com/datasets/andrewmvd/animal-faces
2. **Unsupervised Learning Overview:**
   https://biztechmagazine.com/article/2025/05/what-are-benefits-unsupervised-machine-learning-and-clustering-perfcon
3. **Applications in Diverse Domains:**
   https://pmc.ncbi.nlm.nih.gov/articles/PMC7983091/
4. **Data Exploration and Pattern Discovery:**
   https://analyticalsciencejournals.onlinelibrary.wiley.com/doi/pdfdirect/10.1002/mas.21602
5. **Computer Vision Overview:**
   https://viso.ai/deep-learning/supervised-vs-unsupervised-learning/
6. **SimCLR Paper (Self-Supervised Learning):**
   https://arxiv.org/abs/2002.05709
7. **Unsupervised Learning in NLP:**
   https://milvus.io/ai-quick-reference/what-is-the-role-of-unsupervised-learning-in-nlp
8. **Word2Vec Paper:**
   https://arxiv.org/abs/1301.3781
9. **Latent Dirichlet Allocation (LDA) Paper:**
   https://jmlr.org/papers/v3/blei03a.html
10. **Healthcare and Biomedical Applications:**
    https://pubmed.ncbi.nlm.nih.gov/31891765/
11. **Autonomous Systems and Robotics:**
    https://fiveable.me/introduction-autonomous-robots/unit-7/unsupervised-learning/study-guide/rNorV1tsC0TeCPOO
12. **Recommender and Personalization Systems:**
    https://www.mdpi.com/2073-8994/12/2/185
13. **t-SNE Algorithm:**
    https://lvdmaaten.github.io/tsne/
14. **PCA (Scikit-learn Implementation):**
    https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

15. **K-Means Clustering:**
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

16. **DBSCAN Clustering:**
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

17. **Agglomerative Clustering:**
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html

18. **Adjusted Rand Index (ARI):**
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.adjusted_rand_score.html

19. **Normalized Mutual Information (NMI):**
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.normalized_mutual_info_score.html

20. **Understanding Digital Images for Image Processing and Computer Vision (Medium):**
https://medium.com/%40md-jewel/understanding-digital-images-for-image-processing-and-computer-vision-part-1-cc42be78cca1

21. **RGB Color Model — Wikipedia:**
https://en.wikipedia.org/wiki/RGB_color_model

22. **Image Abstractions — MIT Computational Thinking:**
https://computationalthinking.mit.edu/Fall22/images_abstractions/images/

23. **Understanding Image Data Representation in Computer Systems (DEV Community):**
https://dev.to/adityabhuyan/understanding-image-data-representation-in-computer-systems-4kdm

24. **Digital Image Processing — Wikipedia:**
https://en.wikipedia.org/wiki/Digital_image_processing

25. **Digital Image Processing — Brightness and Contrast (TutorialsPoint):**
https://www.tutorialspoint.com/dip/brightness_and_contrast.htm

26. **Grayscale — Wikipedia:**
https://en.wikipedia.org/wiki/Grayscale

27. **OpenCV Documentation — Color Conversions:**
https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html

28. **Stack Overflow Discussion on Luminance:**
https://stackoverflow.com/questions/596216/formula-to-determine-perceived-brightness-of-rgb-color

29. **Edge Detection using OpenCV (Official Blog):**
https://opencv.org/blog/edge-detection-using-opencv/

30. **Edge Detection - LearnOpenCV:**
https://learnopencv.com/edge-detection-using-opencv/

31. **Wikipedia - Convolutional Neural Network:**
https://en.wikipedia.org/wiki/Convolutional_neural_network

32. **Learn OpenCV - Understanding Convolutional Neural Networks:**
https://learnopencv.com/understanding-convolutional-neural-networks-cnn/

33. **Medium Article on Convolutional Neural Networks:**
https://medium.com/thedeephub/convolutional-neural-networks-a-comprehensive-guide-5cc0b5eae175

34. **Advanced CNN Architectures – Comparative Study:**
https://www.researchgate.net/publication/384631977_CNN_Architectures_for_Image_Classification_A_Comparative_Study_Using_ResNet50V2_ResNet152V2_InceptionV3_Xception_and_MobileNetV2

35. **Deep Learning Architecture Advancements:**
https://thesai.org/Downloads/Volume15No8/Paper_114-Advancements_in_Deep_Learning_Architectures_for_Image_Recognition.pdf