Clean Image Classifier v1 - Documentation

Overview

Clean_Image_v1.py is an advanced, automated image classification and organization tool. It uses computer vision techniques to analyze images and sort them into categories based on similarity to a set of "Reference Images".

Unlike simple classifiers that might just look at file names or basic colors, this script performs a deep analysis of the image content (shapes, textures, edges, colors) to make intelligent decisions. It also "learns" the appropriate strictness for each category by analyzing the consistency of the reference images.

Table of Contents

Prerequisites

The script requires Python and the following libraries: * **OpenCV (`cv2`)**: For advanced image processing (edge detection, histograms). * **Pillow (`PIL`)**: For basic image handling and statistics. * **NumPy (`numpy`)**: For numerical calculations and matrix operations.

You can install them via pip:

```
pip install opencv-python pillow numpy
```

Folder Structure

To use this script, your workspace should be set up as follows:

- `Clean_Image_v1.py`: The script itself.
- `Reference_Images/`: A folder containing subfolders for each category you want to classify (e.g., `dog/`, `cat/`, `car/`). Put 1-5 good example images in each subfolder.

- `animals/` (or your input folder): The folder containing the messy/unsorted images you want to classify.
- `classified_images/`: The script will create this folder and place the organized images here.

How to Run

1. Ensure your `Reference_Images` folder is populated with categories and example images.
2. Ensure your input folder (default is `animals`) contains the images you want to sort.
3. Run the script: `bash python Clean_Image_v1.py`
4. Check the `classified_images` folder for the results.

Detailed Technical Explanation

This section breaks down exactly how the code works, module by module.

1. Feature Extraction

Function: `_extract_enhanced_features(image_path)`

Instead of looking at the image as a whole, the code breaks it down into mathematical "features". For every image, it calculates:

- **Geometry**:
  - **Aspect Ratio**: Is the image tall (portrait) or wide (landscape)?
  - **Area**: How many pixels are in the image?
- **Color Statistics**:
  - **Mean RGB**: The average Red, Green, and Blue values.
  - **Brightness & Contrast**: How light or dark the image is, and the range of tonal difference.
  - **HSV Analysis**: Converts color to Hue (color type), Saturation (intensity), and Value (brightness) for more human-like color perception.
  - **Color Histograms**: Creates a "fingerprint" of the color distribution for R, G, and B channels.
- **Structure & Texture**:
  - **Edge Detection (Canny)**: Detects outlines and boundaries. It calculates "Edge Density" to see if the image is simple (few edges) or complex (many edges).
  - **Texture (Laplacian)**: Measures the variance in pixel intensity to determine if the surface is smooth or rough.

- **Corner Detection**: Counts specific points of interest (corners) to gauge geometric complexity.
- **Binary Ratio**: Converts the image to black and white (using Otsu's method) and calculates the ratio of white to black pixels.

2. Similarity Calculation

Function: `_calculate_advanced_similarity(features1, features2)`

When comparing an unknown image to a reference image, the script doesn't just say "yes" or "no". It calculates a **Weighted Score** (0.0 to 1.0). Different features are more important than others:

- **Edge Similarity (Weight: 3.0)**: The most important factor. If the outlines match, it's likely the same object.
- **Color Similarity (Weight: 2.5)**: Very important. Checks if the overall color palette matches.
- **Corner Density (Weight: 2.5)**: Checks if the complexity of shapes matches.
- **Histogram/Binary/Geometry (Weights: 1.5 - 2.0)**: Secondary checks to refine the match.

The final score is a weighted average of all these comparisons.

3. Dynamic Thresholding

Function: `_calculate_dynamic_thresholds()`

This is the "smart" part of the code. It doesn't use a fixed pass/fail score (like 50%) for every category. 1. It looks at the **Reference Images** for a specific category (e.g., "Tiger"). 2. It compares the reference tigers to *each other*. 3. **High Consistency**: If all reference tigers look very similar, the system sets a **High Threshold** (e.g., 0.85). It learns that "Tigers look very specific, so don't accept anything that isn't a close match." 4. **Low Consistency**: If the reference images vary (e.g., a white tiger and an orange tiger), the system sets a **Lower Threshold** (e.g., 0.65) to be more lenient.

4. Classification Logic

Function: `classify_image(image_path)`

For every input image: 1. **Extract Features**: Run the extraction process described in step 1. 2. **Compare**: Compare these features against *every single reference image* in the library. 3. **Score**: Calculate the similarity score for each category. 4. **Validate**: * The image is assigned to the category with the **Highest Score**. * **Crucial Step**: The score must be higher than the **Dynamic Threshold** for that category. * If the score is too low (below threshold), the image is marked as `unknown`.

5. File Organization

Function: `classify_and_organize_images(...)`

This is the main workflow manager: 1. **Discovery**: It recursively scans the input folder (using `os.walk`) to find all images, even in subfolders. 2. **Processing**: It runs the classification logic on each image. 3. **Copying**: * It copies the image to `classified_images/<Category Name>/`. * **Duplicate Handling**: If a file with the same name already exists, it appends a number (e.g., `image_0.jpg`) to prevent overwriting. 4. **Logging**: It prints detailed logs to the console, showing the score, the threshold used, and the confidence level for every single image.

Step-by-Step Example: Classifying a "Tiger"

To understand exactly how the math works, let's walk through a real scenario: classifying an unknown image (`unknown_01.jpg`) to see if it is a **Tiger**.

1. Reference Collection (The "Learning" Phase)

- **Input**: You place 3 images of tigers in `Reference_Images/tiger/`.
- **Process**: The script starts up. It reads these 3 images and extracts their "Digital DNA" (Features).
- **Result**: It learns that "Tigers" typically have:
  - **Color**: High Orange/Black/White values (Mean RGB).
  - **Texture**: High variance (due to stripes).
  - **Edges**: High edge density (stripes create many edges).
- **Thresholding**: It compares the 3 reference tigers to each other. They look very similar (Consistency = 0.92). The script sets a strict threshold: **0.80**. Any new image must be at least 80% similar to be called a Tiger.

2. Feature Extraction (The "Reading" Phase)

The script looks at `unknown_01.jpg`. It doesn't "see" a cat; it calculates numbers: * **Aspect Ratio**: 1.5 (Landscape). * **Dominant Color (Mean RGB)**: `[50, 100, 200]` (Blue-ish). * **Edge Density**: 0.05 (Very smooth, few edges). * **Texture**: Low variance.

3. Comparison (The "Matching" Phase)

The script compares `unknown_01.jpg` features against the "Tiger" profile.

- **Color Check (Weight 2.5)**:
  - Tiger Profile: Orange `[200, 150, 50]`
  - Unknown Image: Blue `[50, 100, 200]`
  - **Score**: 0.1 (Very low match)
- **Edge/Texture Check (Weight 3.0)**:
  - Tiger Profile: High density (stripes)

  - Unknown Image: Low density (smooth)
  - **Score**: 0.2 (Low match)
- **Geometry Check (Weight 2.0)**:
  - Both are landscape images.
  - **Score**: 0.9 (High match)

Weighted Average Calculation: The script combines these scores based on importance. Since Color and Edges have higher weights than Geometry, the low scores pull the average down. * **Final Similarity Score**: **0.25**

4. The Decision

- **Category**: Tiger
- **Score**: 0.25
- **Required Threshold**: 0.80
- **Result**: `0.25 < 0.80`. **FAIL**.

The script repeats this for every other category (Bear, Wolf, etc.). If the image scores **0.85** against the "Ocean" category (matching the blue color and smooth texture), and the Ocean threshold is 0.70: * **Result**: `0.85 > 0.70`. **PASS**. * **Action**: The image is moved to `classified_images/ocean/`.

If it fails *all* categories, it goes to `classified_images/unknown/`.

Configuration

You can adjust the main settings at the bottom of the script in the `if __name__ == "__main__":` block:

```
IMAGE_FOLDER = "animals"           # Folder containing images to sort
REFERENCE_FOLDER = "Reference_Images" # Folder containing category examples
OUTPUT_FOLDER = "classified_images"   # Where to put the results
```

You can also adjust the default sensitivity in `get_categories_from_reference_folder`: * `default_min=65`: The absolute minimum similarity score (65%) required to match a category. * `default_max=100`: The maximum threshold cap.