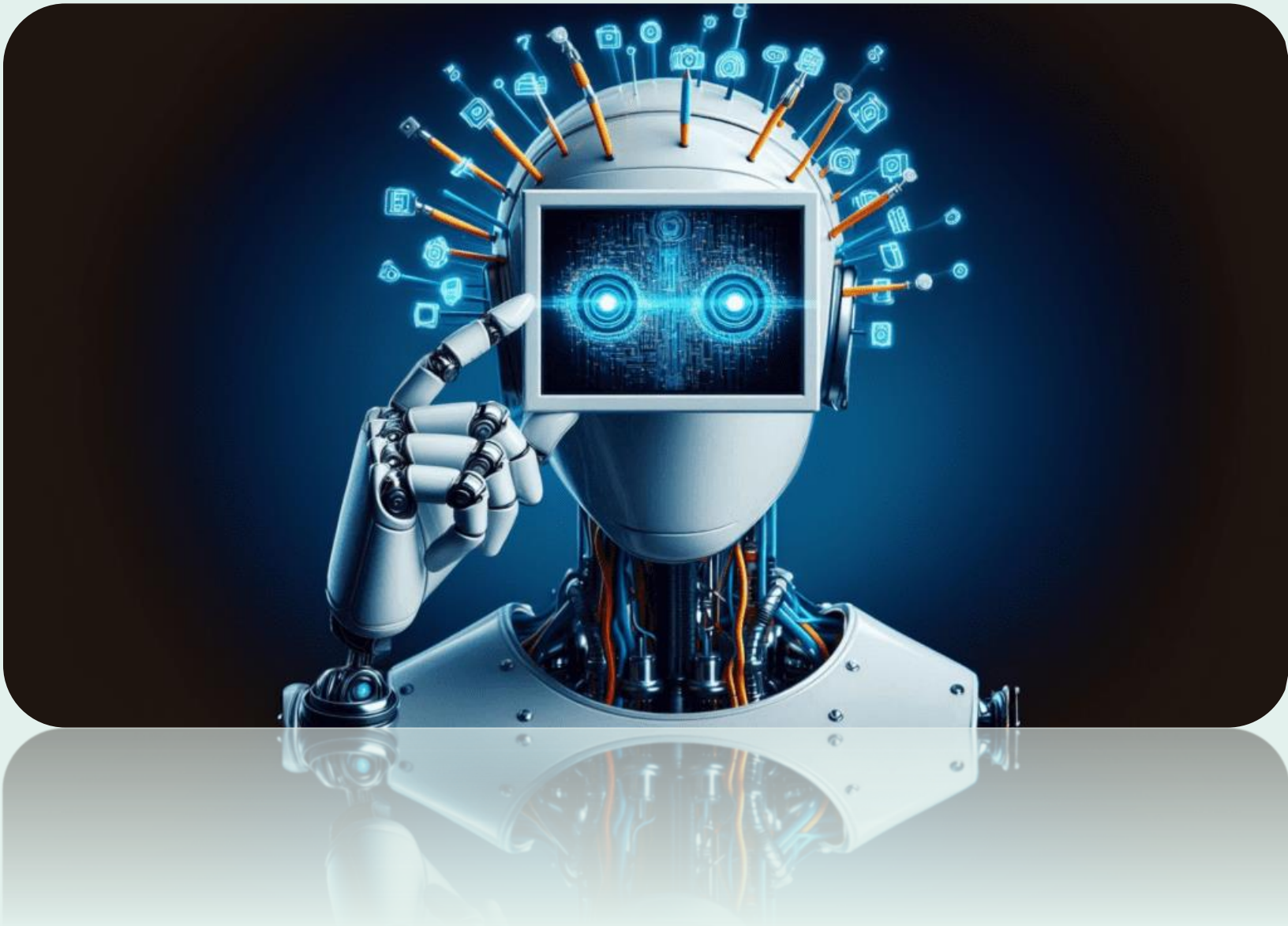
CSE480 : Machine Vision



Supervised by:

Dr. Hossam Hassan

Eng. Dina

MCTA

Team Members	IDs
Abdelrahman Ahmed Emam	20P9733
Mariam Medhat Sadek FARAGALLAH	20P8306

Table of Contents

Problem Definition and Importance

Image classification is a fundamental task in computer vision, involving the automatic assignment of labels to images based on their visual content. This process is critical in various real-world applications such as facial recognition, medical diagnosis, and autonomous vehicles. The need for accurate and efficient image classification methods has driven extensive research and technological advancements. This research aims to explore and implement different techniques for image classification, emphasizing feature extraction methods and traditional classifiers to evaluate their performance.

The objective of this work is to classify RGB images into different categories using different feature extraction techniques and traditional classifiers. Feature extraction is a crucial step that transforms raw image data into a structured format suitable for classification. Three specific feature extraction methods are considered: Histogram of Oriented Gradients (HOG), color histograms, and Local Binary Patterns (LBP). These methods capture various aspects of image content such as texture, color distribution, and edge orientations, which are essential for distinguishing between image categories.

The chosen feature extraction techniques have been selected due to their proven effectiveness in prior research and their interpretability in visual pattern recognition tasks. HOG features are commonly used in detecting objects by focusing on edge directions, while color histograms summarize color distributions, making them suitable for distinguishing images based on color-based features. LBP histograms provide valuable texture information, enabling robust classification even in challenging scenarios involving varying lighting conditions.

To classify the extracted feature vectors, traditional classifiers such as K-Nearest Neighbours (KNN), K-Means clustering, and Support Vector Machines (SVM) are employed. KNN is a straightforward and intuitive classifier that assigns labels based on the majority class of the nearest neighbours. K-Means clustering, on the other hand, performs unsupervised learning by grouping similar data points into clusters. Lastly, SVM is a powerful and well-established supervised learning algorithm that finds the optimal decision boundary between classes.

The decision to use these classifiers stems from their complementary characteristics and widespread application in previous research. Implementing KNN and K-Means from scratch encourages a deeper understanding of their underlying mechanisms, while leveraging the Scikit-learn library for SVM ensures a reliable and scalable implementation. This hybrid approach balances learning, implementation complexity, and performance evaluation.

The significance of this research lies in its comprehensive exploration of various image classification techniques. By comparing the performance of different feature extraction methods and classifiers, valuable insights into their strengths and limitations can be obtained. This analysis contributes to the broader field of computer vision by highlighting effective combinations of features and classifiers, potentially guiding future research and practical applications.

Overall, this study emphasizes the importance of selecting appropriate feature extraction techniques and classifiers for image classification tasks. It also underscores the critical role of research-driven experimentation in advancing computer vision technologies and developing more accurate and efficient image classification systems

Feature Extraction Methods

I. Histogram of Oriented Gradients (HOG)

The ***Histogram of Oriented Gradients (HOG)*** is a feature descriptor used primarily in computer vision and image processing for tasks like object detection, especially pedestrian (human) detection. It captures edge and gradient structures, making it highly effective for identifying shapes and objects in images.

Concept of HOG

A histogram is an approximate representation of the distribution of numerical data that looks like a looks a bar graph. Each bar represents a group of data that falls in a certain range of values, also called bins. Orientation means the direction or orientation of an image gradient. HOG will produce a histogram of gradient directions in an image, so it works by analyzing the gradients (changes in pixel intensities) within an image. It focuses on the shape and structure of objects rather than specific pixel values, making it robust against changes in lighting and appearance.

Steps in Computing HOG

1. **Preprocessing the Image**
    - Convert the image to grayscale (if it's not already), as HOG operates on intensity changes.
    - Optionally apply normalization techniques to reduce the effect of lighting variations.
2. **Gradient Computation**
    - Compute gradients in the x and y directions using filters like Sobel operators.
    - Calculate: G_x = I (x + 1, y) – I (x−1, y)
      G_y = I (x, y +1) – I (x, y - 1)
    - Determine the gradient magnitude MMM and orientation θ :

$$M = \sqrt{{G_x}^2 + {G_y}^2}$$

$$\theta = tan^{-1}(\frac{G_y}{G_x})$$

3. **Divide the Image into Cells**

    Split the image into small, non-overlapping blocks called **cells** (e.g., 8x8 pixels).

4. **Histogram Generation**
    - For each cell, create a histogram of gradient directions.
    - The orientation range is typically divided into bins (e.g., 9 bins from 0° to 180° for unsigned gradients or 0° to 360° for signed gradients).
    - Pixels contribute to the bins based on their gradient magnitudes.

5. **Block Normalization**
    - To handle illumination variations, normalize the histograms across larger overlapping blocks (e.g., 16x16 pixels consisting of four 8x8 cells).
    - Common normalization methods include:
        - **L2-Norm**: $\frac{v}{\sqrt{||v||_2^2 + \epsilon^2}}$
        - **L1-Norm**: $\frac{v}{||v||_1 + \epsilon}$
    - This step ensures better invariance to lighting and contrast changes.
6. **Feature Vector Creation**
    - Concatenate all normalized histograms into a single feature vector. This vector becomes the HOG descriptor for the entire image or region of interest.

Applications of HOG

1. **Object Detection**: Widely used in pedestrian and vehicle detection.
2. **Face Detection**: Utilized in facial recognition systems.
3. **Image Classification**: Used in various classification models.
4. **Medical Imaging**: Applied in medical diagnosis tasks.

Advantages of HOG

- **Robust to Lighting Changes**: Gradient-based features are less sensitive to lighting variations.
- **Shape Recognition**: Excellent for detecting object outlines and shapes.
- **Rotation Invariance (to some extent)**: Can handle small rotations due to block normalization.

Limitations of HOG

- **Computational Cost**: HOG can be resource-intensive, especially for large images.
- **Not Fully Rotation-Invariant**: Significant rotations can reduce effectiveness.
- **Fixed Window Size**: Requires careful tuning of cell and block sizes for different tasks.

II. Local Binary Patterns(LBP)

Local Binary Patterns (LBP) is a texture descriptor widely used in computer vision for tasks such as face recognition, texture classification, and object detection. It efficiently captures local spatial patterns by comparing pixel intensities, making it highly effective for distinguishing textures.

Concept of LBP

LBP encodes the texture of an image by comparing each pixel's intensity with its surrounding pixels. It produces a binary code that represents the local structure, making the feature robust to lighting changes and grayscale transformations.

How LBP Works

1. **Neighborhood Definition**
   - Consider a central pixel in a grayscale image.
   - Define a neighborhood of surrounding pixels (usually 3x3, though larger neighbourhoods can be used).
   - Number these surrounding pixels clockwise, starting from the top-left.
2. **Thresholding**
   - Compare each surrounding pixel's intensity with the central pixel's intensity:
     - If the surrounding pixel's intensity I (x_i) is greater than or equal to the central pixel's intensity I (x_c), assign **1**.
     - Otherwise, assign **0**.
3. **Binary Pattern Creation**
   - After thresholding, form an 8-bit binary number using the results from the neighborhood.
   - Convert this binary number to a decimal value to create the LBP code.
4. **Feature Vector Generation**
   - The computed LBP code replaces the central pixel's value in the resulting image.
   - Create a histogram of LBP codes for the entire image or regions of interest, forming the feature vector.

Mathematical Representation

The LBP value for a central pixel (x_c ,y) is calculated as:

LBP(x_c , y) = $\sum_{i=0}^{P-1} s(I(x_i) - \mathrm{I}(x_c)).2^i$

Where:

- $I(x_i)$: Intensity of the neighboring pixel.
- $\mathrm{I}(x_c)$: Intensity of the central pixel.
- P : Number of neighboring pixels.
- $s(x)$: Threshold Function

$$s(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Types of LBP Variants

1. **Basic LBP**
    - Operates on a 3x3 neighborhood, resulting in 8-bit binary patterns.
2. **Circular LBP (Extended LBP)**
    - Works with circular neighborhoods, allowing flexible pixel counts and radii for larger areas.
    - This improves rotation invariance and texture capture.
3. **Uniform LBP**
    - Only considers patterns with at most two bitwise transitions (e.g., 11100000 has two transitions).
    - Reduces the number of LBP codes, improving computational efficiency.
4. **Rotation-Invariant LBP**
    - Shifts binary patterns to find the smallest possible code, achieving rotational invariance.

Applications of LBP

1. **Face Recognition**: LBP is used in real-time face recognition systems.
2. **Texture Classification**: Distinguishes between various textures in images.
3. **Medical Imaging**: Detects patterns in X-rays, MRIs, and CT scans.
4. **Object Detection**: Identifies textured objects and environments.

Advantages of LBP

- **Simple and Efficient**: Computationally inexpensive.
- **Robust to Lighting Changes**: Works well with varying illumination.
- **Texture and Pattern Analysis**: Effectively captures fine-grained texture details.

Limitations of LBP

- **Noise Sensitivity**: Small pixel intensity variations can cause different LBP codes.
- **Limited Discriminative Power**: Struggles with complex and large-scale textures.
- **Fixed Neighborhood Size**: May miss larger or more distant patterns if the neighborhood size is too small.

III. Color Histogram

Color Histogram is a widely used feature descriptor in computer vision and image processing. It captures the distribution of colours in an image by counting the number of pixels that fall into specific color ranges (bins). This statistical representation is crucial for tasks like image retrieval, classification, and object recognition.

Concept of a Color Histogram

A color histogram represents the frequency of color intensities in an image. It is created by dividing the color space into intervals (bins) and counting how many pixels fall into each bin. This produces a graphical representation showing the number of pixels for each color range.

Steps to Compute a Color Histogram

1. **Choose a Color Space**
    - Common color spaces include:
        - **RGB** (Red, Green, Blue)
        - **HSV** (Hue, Saturation, Value)
        - **Lab** (Lightness, a, b)
        - **Grayscale** (for single-channel intensity)
    - The choice of color space depends on the application (e.g., HSV is better for illumination invariance).
2. **Quantize the Color Space**
    - Divide the chosen color space into discrete bins.
    - For example, in RGB space, dividing each channel into 8 bins results in 8×8×8 = 512 bins.
3. **Count Pixel Values**
    - Iterate through each pixel of the image.
    - Determine the bin corresponding to the pixel's color value.
    - Increment the pixel count for that bin.
4. **Normalize the Histogram (Optional)**
    - Normalize the histogram to make it independent of image size:

$$H'(i) = \frac{H(i)}{N}$$

Where:

- $H(i)$: Number of pixels in bin iii.
- N : Total number of pixels in the image.

Mathematical Representation

Given an image with pixels $p\ (x\ ,y)$, a color histogram H is defined as:

$$H(i) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \delta(B\big(p(x\ ,y)\big) - i)$$

Where:

- M and N : Image dimensions.
- $B\big(p(x\ ,y)\big)$: The bin index for pixel $p\ (x\ ,y)$
- δ : Indicator function, returning 1 if the argument is true, otherwise 0.

Types of Color Histograms

1. **1D Histogram**:
    - Uses a single color channel (e.g., only red, green, or blue).
    - Suitable for grayscale images.
2. **2D Histogram**:
    - Combines two color channels (e.g., hue and saturation in HSV space).
    - Offers better color differentiation.
3. **3D Histogram**:
    - Uses three color channels (e.g., R, G, and B).
    - Provides the most detailed color distribution but at a higher computational cost.

Applications of Color Histograms

1. **Image Retrieval**: Search and match images based on color content.
2. **Object Detection**: Identify objects with unique color patterns.
3. **Video Analysis**: Track coloured objects in video sequences.
4. **Medical Imaging**: Analyze coloured regions in medical images.
5. **Content-based Image Retrieval (CBIR)**: Find similar images in large datasets.

Advantages of Color Histograms

- **Simple and Fast**: Easy to compute and understand.
- **Rotation and Translation Invariant**: Independent of object orientation or position.
- **Global and Local Features**: Can describe entire images or specific regions.

Limitations of Color Histograms

1. **No Spatial Information**: The histogram only captures color distribution, ignoring spatial relationships between pixels.
2. **Lighting Sensitivity**: Results can be affected by changes in illumination and shadows.
3. **Color Quantization Issues**: Too many bins increase computational cost, while too few reduce detail.

Classification Algorithms

I. K- Nearest Neighbors (KNN)

The ***K-Nearest Neighbours (KNN)*** algorithm is a simple, non-parametric, and lazy learning classification method used in machine learning and pattern recognition. It works by finding the 'k' closest data points to a query point and assigning the most common label among them as the prediction.

Key Characteristics of KNN

1. **Instance-based Learning**: KNN memorizes the training dataset, making it a lazy learner.
2. **Non-parametric**: No assumptions are made about the data distribution.
3. **Supervised Learning**: Used for classification and regression tasks.

How KNN Works

Step 1: Data Preprocessing

- Normalize or scale features to ensure fair distance computation.
- Split data into training and testing sets.

Step 2: Choose the Value of 'K'

- 'K' determines how many neighbours to consider.
- Common methods to choose 'K':
  - Use cross-validation to find the optimal value.
  - Use $K = \sqrt{N}$, where N is the number of data points.
  - Select odd values of 'K' to break ties in binary classification.

Step 3: Distance Metric Selection

KNN uses a distance metric to calculate how close data points are. The commonly used metric include

Euclidean Distance Rule :

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

Step 4: Classification or Regression

1. **Classification (Majority Voting)**:
   - Identify the 'K' nearest neighbors.
   - Count the labels of these neighbors.
   - Assign the class with the highest frequency (majority voting).
2. **Regression (Averaging Neighbors)**:
   - Predict the target value by averaging the values of the nearest neighbors.

Step 5: Model Prediction

- The test data point is classified based on the calculated majority class or predicted regression value.

Step 6: Model Evaluation

- Evaluate the model using metrics such as:
    - **Accuracy** (for classification)
    - **Mean Squared Error (MSE)** (for regression)
    - **Confusion Matrix**, **Precision**, **Recall**, and **F1-score** for detailed classification performance.

Example of KNN Classification

Imagine a dataset with two features: height and weight, where we classify people as either "Athlete" or "Non-Athlete."

1. A new person with height = 170 cm and weight = 65 kg needs classification.
2. Use Euclidean distance to calculate the distance between this point and every point in the dataset.
3. Suppose 'K' = 3. Find the three closest points.
4. If two points are labelled "Athlete" and one is "Non-Athlete," the person is classified as "Athlete."

Advantages of KNN

1. **Simple to Understand and Implement**: Requires minimal assumptions about the data.
2. **No Training Phase**: Since it's a lazy learner, training is virtually instantaneous.
3. **Effective with Multiclass Problems**: Easily adaptable to multiple classes.
4. **Robust with Sufficient Data**: Performs well when large, clean datasets are available.

Limitations of KNN

1. **Computationally Expensive**: KNN requires distance computation for every query, which is costly for large datasets.
2. **Memory-Intensive**: Must store the entire training set in memory.
3. **Sensitive to Noise**: Outliers can heavily affect predictions.
4. **Curse of Dimensionality**: Performance degrades with increasing feature dimensions, as distances become less meaningful.

Applications of KNN

1. **Image Recognition**: Classifying objects and faces.
2. **Medical Diagnosis**: Identifying diseases based on symptoms.
3. **Recommendation Systems**: Finding similar users or items.
4. **Financial Analysis**: Detecting credit card fraud and predicting stock market trends.

Improving KNN Performance

1. **Feature Scaling**: Standardize features to avoid dominance by higher-range features.
2. **Dimensionality Reduction**: Use techniques like PCA to reduce feature space.
3. **Weighted Voting**: Assign higher weights to closer neighbors.
4. **Optimized Search Algorithms**: Use KD-Trees or Ball Trees for faster searches in large datasets.

II. K-Means

K-Means is an unsupervised machine learning algorithm used for clustering tasks. It groups data into *k* clusters based on feature similarity, minimizing the variance within each cluster. Unlike classifiers, K-Means does not require labelled data, making it suitable for tasks like customer segmentation, image compression, and anomaly detection.

How K-Means Works

Step 1: Initialization

- Choose the number of clusters k.
- Randomly initialize k centroids (cluster centers) from the data points.

Step 2: Assignment of Data Points

- For each data point x_i, calculate the distance between the point and each centroid.
- Assign the point to the nearest centroid's cluster using a distance metric, commonly **Euclidean distance**: $d(x_j, C_j) = \sqrt{\sum_{m=1}^{n}(x_{im} - C_{jm})^2}$

Where:

- x_{im} : Value of the feature of data point x_i
- C_{jm}: Value of the $m-th$ feature of centroid C_j
- n : Number of features

Step 3: Update Centroids

- Recalculate the centroids by computing the mean of all points assigned to each cluster:

$$C_j = \frac{1}{N_j} \sum_{i \in Cluster_j} x_i$$

Where:

- C_j : New centroid for cluster j
- N_j : Number of points in cluster j

Step 4: Repeat

- Repeat Steps 2 and 3 until convergence, meaning the centroids no longer change or the maximum number of iterations is reached.

Stopping Criteria

1. Centroids stabilize (no significant change).
2. Maximum number of iterations reached.
3. Minimum improvement in the cost function.

Cost Function (Objective Function)

K-Means minimizes the sum of squared distances between points and their centroids:

$$J = \sum_{j=1}^{k} \sum_{i \in Cluster_j} ||x_i - C_j||^2$$

Choosing the Number of Clusters (k)

1. **Elbow Method**:
    - Plot the cost function (inertia) versus different values of k.
    - Look for the "elbow point," where the reduction in cost starts diminishing.
2. **Silhouette Score**:
    - Measures how similar a point is to points in its cluster compared to points in other clusters.
    - A higher score indicates better clustering.
3. **Gap Statistic**:
    - Compares clustering performance with expected performance from randomly generated data.

Distance Metrics Used in K-Means

1. **Euclidean Distance** (The one we used):
    - Assumes spherical clusters with equal variance.
2. **Manhattan Distance**:
    - Suitable for high-dimensional data with different feature ranges.
3. **Cosine Similarity**:
    - Useful for text data or non-Euclidean spaces.

Example of K-Means Clustering

Suppose we have a dataset of customers with two features: **Annual Income** and **Spending Score**.

1. Choose $k = 3$.
2. Randomly initialize 3 centroids.
3. Assign each customer to the nearest centroid.
4. Recalculate centroids based on assigned points.
5. Repeat the process until centroids stabilize.
6. Interpret the clusters as different customer segments (e.g., high spenders, low spenders).

Applications of K-Means

1. **Customer Segmentation**: Grouping customers based on buying behavior.
2. **Image Compression**: Reducing image sizes by clustering pixel colors.
3. **Document Clustering**: Organizing similar text documents or articles.
4. **Anomaly Detection**: Identifying unusual data points as outliers.
5. **Market Segmentation**: Grouping products based on sales or features.

Advantages of K-Means

1. **Simple and Easy to Implement**: Requires minimal setup.
2. **Scalable**: Works well with large datasets when properly optimized.
3. **Efficient**: Has a time complexity of $O(n * k * i)$, where n is the number of data points, k is the number of clusters, and iii is the number of iterations.

Limitations of K-Means

1. **Requires Predefined k**: The number of clusters must be specified in advance.
2. **Sensitive to Initialization**: Random centroids can lead to different results.
3. **Assumes Spherical Clusters**: Works poorly with non-spherical clusters.
4. **Sensitive to Outliers**: Outliers can skew centroid calculations.
5. **Not Suitable for Categorical Data**: Works best with numerical data.

Improving K-Means Performance

1. **Multiple Initializations**: Use **KMeans++** initialization to select better starting centroids.
2. **Data Preprocessing**: Normalize or scale data to ensure fair distance computations.
3. **Dimensionality Reduction**: Apply PCA to reduce data dimensions.
4. **Use Alternative Algorithms**: Consider hierarchical or density-based clustering (DBSCAN) if data has complex structures.

III. Support Vector Machine (SVM)

Support Vector Machine (SVM) is a powerful supervised learning algorithm used for classification and regression tasks. It works by finding the optimal hyperplane that separates data points into different classes while maximizing the margin between the closest points from each class.

Key Concepts of SVM

1. **Hyperplane**: A decision boundary that separates different classes. In a 2D space, it's a line, while in higher dimensions, it becomes a plane or a hyperplane.
2. **Support Vectors**: Data points closest to the hyperplane, which influence its position.
3. **Margin**: The distance between the hyperplane and the nearest data points from either class. SVM maximizes this margin for better generalization.

How SVM Works

Step 1: Data Preprocessing

- Standardize or normalize the data to ensure fair distance computation.
- Split data into training and testing sets.

Step 2: Finding the Optimal Hyperplane

SVM finds a hyperplane that maximizes the margin between classes by solving the following optimization problem: $Maximize\ M = \frac{2}{||\omega||}$

Where:

- M : Margin width
- ω : Weight vector (parameters of the hyperplane)

The hyperplane is represented by:

$$\omega . x + b = 0$$

Where:

- ω : Normal vector to the hyperplane
- x : Input data points
- b: Bias term

The decision boundary conditions are:

$$\omega . x_i + b \geq +1\ for\ y_i = +1$$

$$\omega . x_i + b \geq -1\ for\ y_i = -1$$

Thus, SVM aims to maximize the margin by minimizing $||\omega||^2$ subject to these constraints.

Step 3: Handling Non-linearly Separable Data

In real-world scenarios, data is often non-linearly separable. SVM addresses this using:

1. Kernel Trick

- SVM applies a kernel function to transform data into a higher-dimensional feature space where it becomes linearly separable.

Common kernel functions:

- **Linear Kernel**: $K(x_i, x_j) = x_i . x_j$
- **Polynomial Kernel**: $K(x_i, x_j) = (x_i . x_j + c)^d$
- **Radial Basis Function (RBF) Kernel (Gaussian)**: $K(x_i, x_j) = exp(-\gamma ||x_i - x_j||^2)$
- **Sigmoid Kernel**: $K(x_i, x_j) = \tanh(\alpha x_i . x_j + c)$

2. Soft Margin SVM

- Introduces a slack variable ε_i to allow some misclassifications while minimizing an additional penalty.
- The new objective becomes: $\min \frac{1}{2} ||\omega||^2 + C \sum_{i=1}^{N} \varepsilon_i$

Where:

- C: Regularization parameter controlling the trade-off between maximizing the margin and minimizing misclassification errors.

Mathematical Representation of SVM Classifier

The SVM decision function is given by: $f(x) = sign(\omega . \varphi(x) + b)$

Where:

- $\varphi(x)$: Transformation of the input using a kernel function
- b : Bias term

Example of SVM Classification

Consider a dataset with two features: **Exam Score** and **Study Hours**.

1. The task is to classify students as "Pass" or "Fail."
2. SVM finds the best hyperplane that separates these two classes.
3. If the data points are not linearly separable, an RBF kernel is applied.
4. Support vectors define the boundary, and the algorithm classifies new students based on which side of the hyperplane they fall.

Applications of SVM

1. **Image Classification**: Recognizing objects, faces, or handwritten digits.
2. **Text Categorization**: Sorting documents into categories (e.g., spam vs. non-spam).
3. **Sentiment Analysis**: Classifying text into positive or negative sentiment.
4. **Medical Diagnosis**: Detecting diseases based on patient data.
5. **Bioinformatics**: Gene and protein classification tasks.

Advantages of SVM

1. **Effective in High-Dimensional Spaces**: Works well even with a large number of features.
2. **Robust to Overfitting**: Especially effective with a properly tuned kernel and regularization.
3. **Versatile**: Works for linear and non-linear data.
4. **Memory-Efficient**: Only support vectors are stored for prediction.

Limitations of SVM

1. **Sensitive to Kernel Choice**: Requires proper kernel and hyperparameter tuning.
2. **Not Suitable for Large Datasets**: Training time increases with data size.
3. **No Probabilistic Interpretation**: SVM does not directly provide probability estimates.
4. **Sensitive to Noisy Data and Outliers**: Can be affected if outliers are not handled.

SVM Model Evaluation Metrics

- **Accuracy**: Measures overall performance.
- **Precision**: Measures correctness of positive predictions.
- **Recall (Sensitivity)**: Measures ability to detect positive instances.
- **F1 Score**: Harmonic mean of precision and recall.
- **Confusion Matrix**: Provides a breakdown of classification results.

Improving SVM Performance

1. **Data Preprocessing**: Normalize or scale features to ensure fair distance computation.
2. **Kernel Tuning**: Experiment with different kernel functions and parameters.
3. **Grid Search**: Use grid search with cross-validation to optimize hyperparameters C , γ, and kernel types.
4. **Remove Outliers**: Preprocess data to remove noisy data points.
5. **Dimensionality Reduction**: Apply PCA or feature selection methods.

Experimental Results and Discussions

Comments on HOG :

```
Train HOG features shape: (3060, 8100)
Test HOG features shape: (6084, 8100)
```

For the training dataset, we have 3060 images. For each image the number of features extracted was 8100 features.

A dimensionality reduction of 50 was applied which means that the number of features extracted for each of the 3060 images will be equal to 50.

The same was done to the testing dataset but the number of images is 6084.

Comments on LBP:

```
Train LBP features shape: (3060, 202)
Test LBP features shape: (6084, 202)
```

For the training dataset, we have 3060 images. For each image the number of features extracted was 202 features.

A dimensionality reduction of 50 was applied which means that the number of features extracted for each of the 3060 images will be equal to 50.

The same was done to the testing dataset but the number of images is 6084.

Comments on Color Histogram :

```
Train Color Histogram features shape: (3060, 24576)
Test Color Histogram features shape: (6084, 24576)
```

For the training dataset, we have 3060 images. For each image the number of features extracted was 24576 features.

A dimensionality reduction of 100 was applied which means that the number of features extracted for each of the 3060 images will be equal to 100.

The same was done to the testing dataset but the number of images is 6084.

Accuracies for the three feature Extractions for the K-Means :

```
HOG RESULTS
Clustering accuracy on training set: 19.98%
Clustering accuracy on validation set: 19.44%
LBP RESULTS
Clustering accuracy on training set: 6.45%
Clustering accuracy on validation set: 10.78%
COLOR RESULTS
Clustering accuracy on training set: 5.19%
Clustering accuracy on validation set: 7.68%
```

Accuracies for the three feature Extractions for the KNN :

```
Accuracy of KNN on validation set: 32.84%
Accuracy of KNN on validation set: 14.05%
Accuracy of KNN on validation set: 13.24%
```

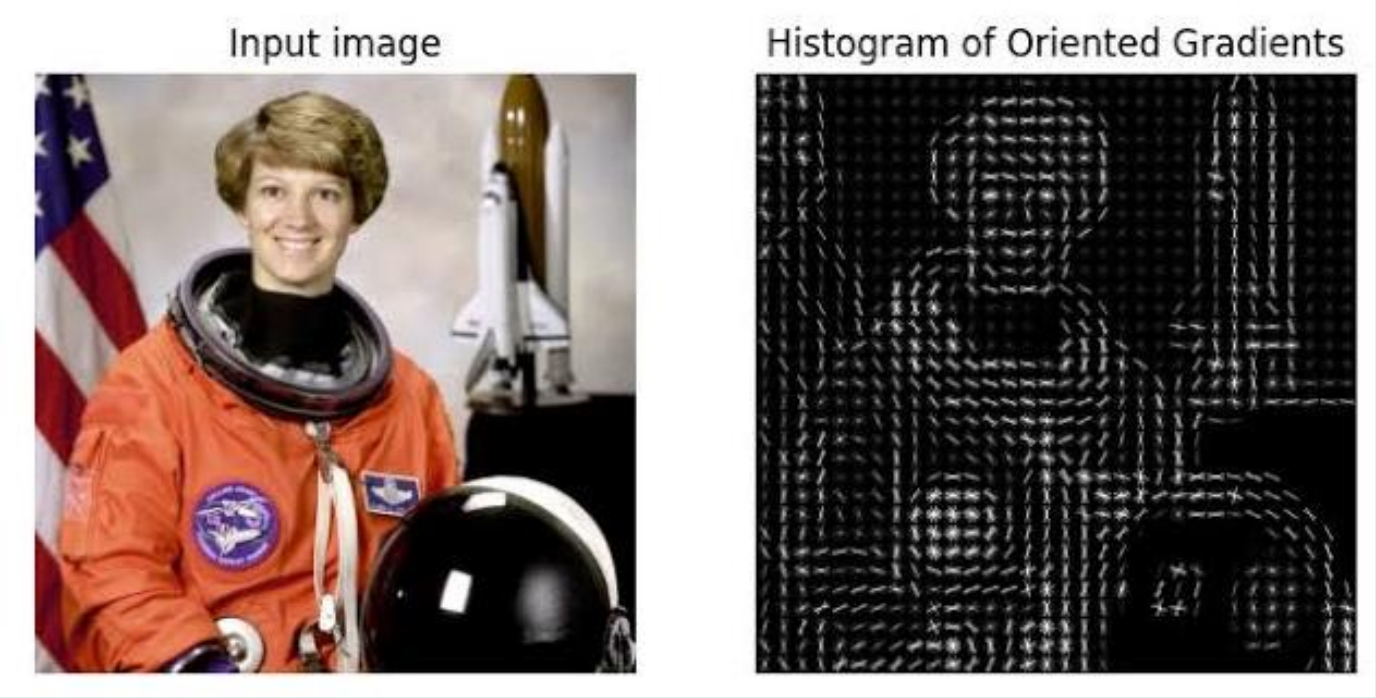
Accuracies for the three feature Extractions for the SVM :

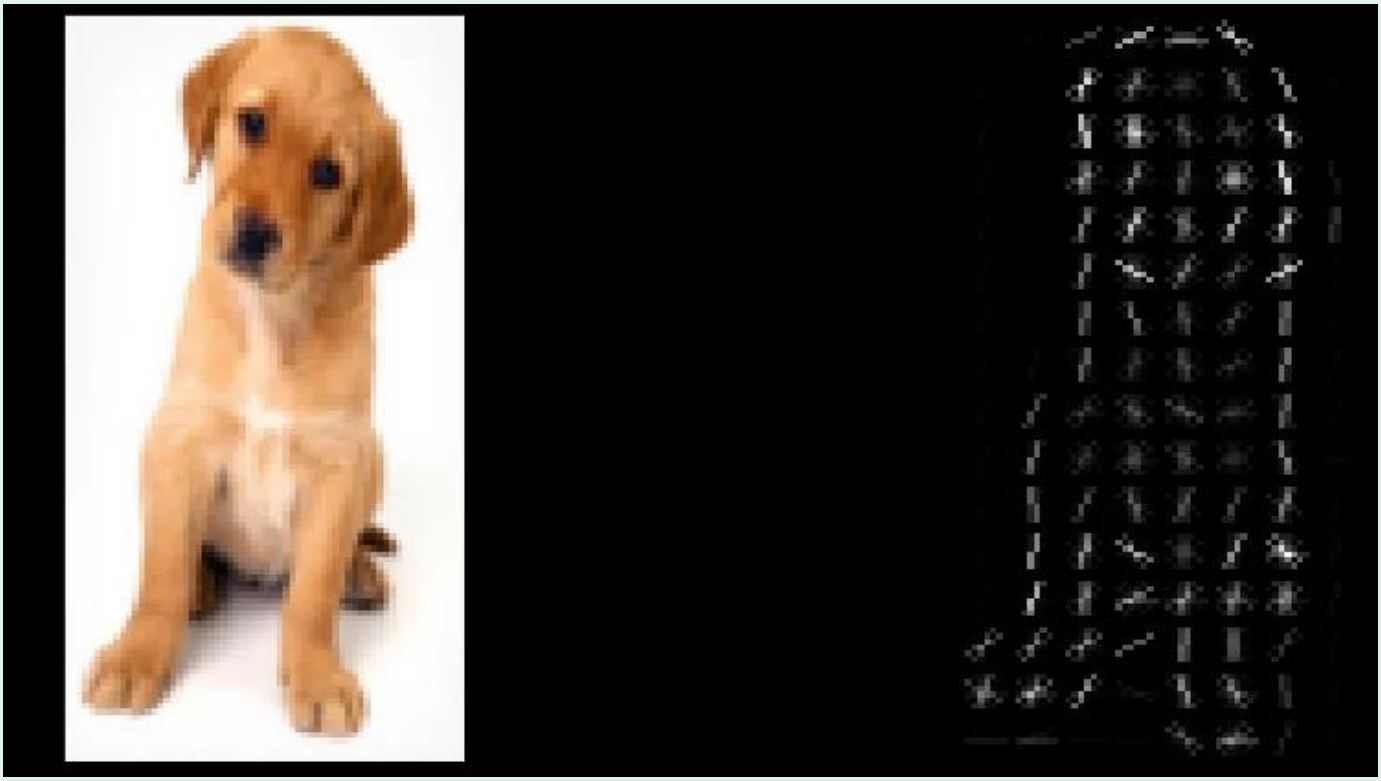
```
Accuracy on test (HOG) set: 60.24%
Accuracy on test (LBP) set: 38.84%
Accuracy on test (color histogram) set: 20.89%
```

General Comments:

First Comment:

HOG has mainly the best accuracy of all three classification methods used.
It focuses on the shape and structure of objects rather than specific pixel values, making it robust against changes in lighting and appearance. It extracts efficient features that can helps in object detection like the figure below. From this image, it detect the edges of objects in images by calculations their gradients while diminishing the gradients value of flat surfaces such as the background to highlight the features of the object more than any other thing.

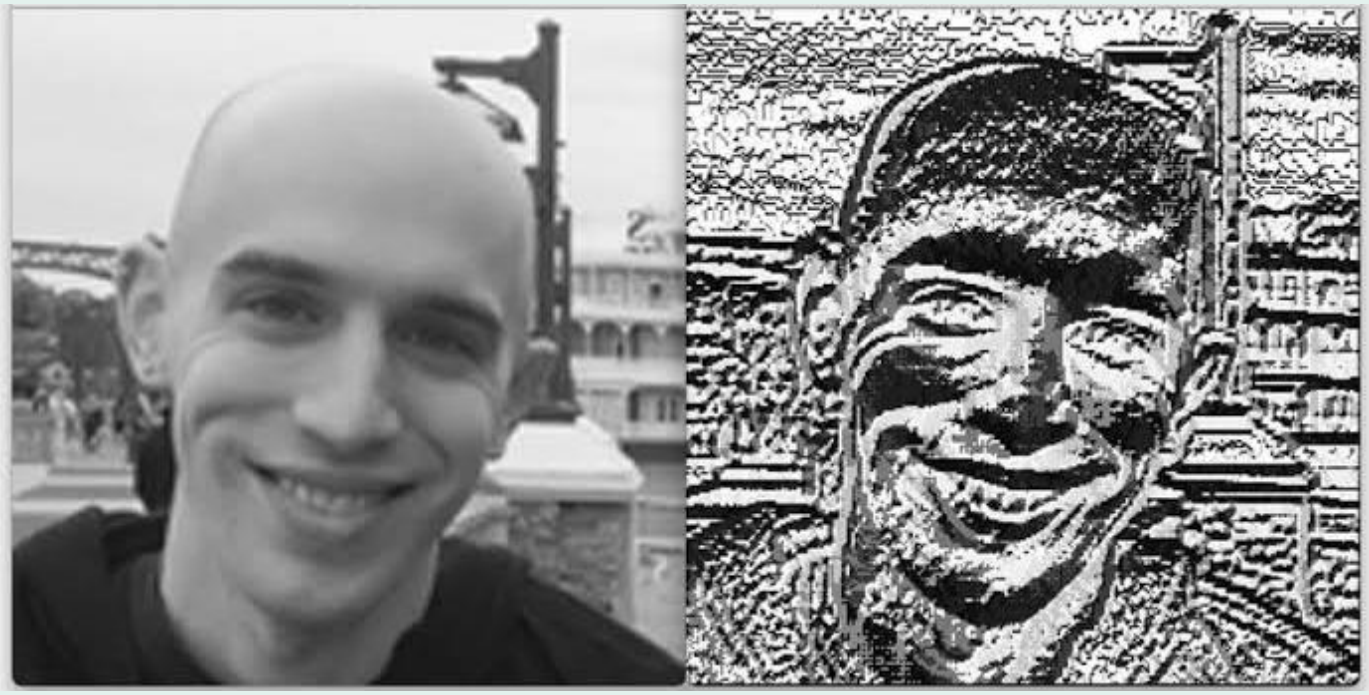




Second Comment:

LBP comes secondly in terms of accuracies. It is still an effective way to detect objects, but we can still find noises especially on flat surfaces which affects some edges, and as a result, the accuracy decreases accordingly.

Such as here in the figure, the forehead of this person elaborated a lot of noise despite it being a single zone with few features, but the noise affected the feature extraction of this part in the image.

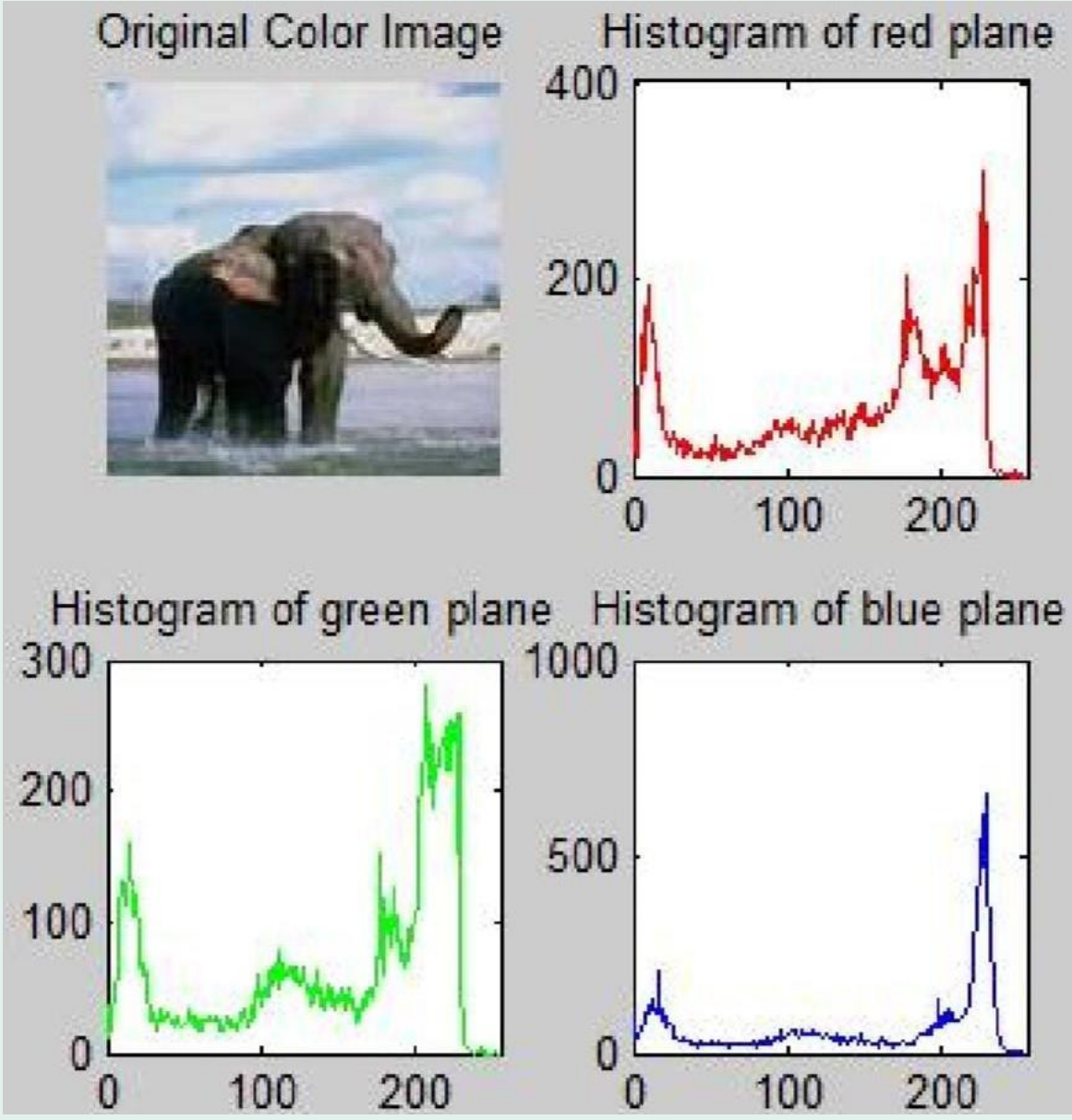


Third Comment:

The Color Histogram has the worst accuracies among the three methods of feature extraction.

As we can see, the extraction is based mainly on the colors in the given image. It generates the histogram of each color of the RBG separately then concatenates them (merges the three colors to have a single output. (Such as the three graphs elaborated from the image of the elephant)

Of course, this flaw affects the feature extraction. Imagine the images of a sky and another one of the a blue sea, if extraction is based only on color, the blue will be the dominant making these two images classified under the same category, so it fails in these cases.



Fourth Comment :

The SVM has given the best accuracy among the three classifiers algorithms.

Why? Because SVM creates a boundary decision for classification as it takes data and uses RBF kernel (radial basis function). This is a non-linear kernel because, as the data is complex, we need to have a non-linear data boundary margin. So, if 2 images have similar features when trying to differ it the margin helps. Example, if we have a cat image and baby cheetah image, both have similar features, so they appear close to each other on graph, so the margin plays crucial rule in differentiating them.

Merging the HOG feature extraction with this classifier (SVM), we get the highest accuracy among other classifiers.

Fifth Comment :

The KNN algorithm gets second place in accuracies as it compares input data with all data by calculating the distance between them, then sorting the distances ascendingly, then comparing top K elements to see each label of them. The most frequent data label from the input data will be the true label for the algorithm even if it is wrong.

Like our example, the cat and baby cheetah have common features. So, if we took k for 5, it might see 3 data labels for the cat and 2 for the baby cheetah. Finally, it will decide that the given image is a cat, while it might be a baby cheetah.

Sixth Comment :

The K-Means is the worst among them because it is based on Clustering. The Clusters are labeled from the classifier depending on K value but it can cluster wrong data if it has similar features again with our cat and baby cheetah example so some centroids may have no actual data, so the error is so high. Misclassification occurs frequently.

Appendix with Codes

Import for libraries

```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
from skimage.feature import hog
from skimage.color import rgb2gray
from sklearn.model_selection import train_test_split
from skimage.feature import local_binary_pattern
import matplotlib.pyplot as plt
from collections import Counter
from sklearn.decomposition import PCA
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from skimage import exposure
import cv2
from sklearn.metrics import accuracy_score, classification_report
```

Data Loading

```
[ ]  dataset, info = tfds.load('caltech101', with_info=True, as_supervised=True)

    # The dataset is divided into training and test splits.
    train_dataset_original_hog = dataset['train']
    test_dataset_original_hog = dataset['test']

    train_dataset_original_lbp = dataset['train']
    test_dataset_original_lbp = dataset['test']

    train_dataset_original_color = dataset['train']
    test_dataset_original_color = dataset['test']
    print("DONE!")
```

What this code does?

- Download dataset with its information and make it suitable for supervised classification methods
- Separate data into training and testing
- Each data is made for the 3 classifiers

Process Data

For HOG :

```
[ ]  IMG_SIZE_HOG = (128, 128)

     def preprocess_image_hog(image, label):
         """
         Resize and normalize the image.
         Args:
             image: The input image.
             label: The label corresponding to the image.
         Returns:
             Resized and normalized image, label.
         """
         # Resize the image
         image = tf.image.resize(image, IMG_SIZE_HOG)
         # Normalize the image to the range [0, 1]
         image = image / 255.0
         return image, label
```

What this code does?

- Re-size each image to 128x128 (Note: bigger sizes reduce accuracy)
- Then normalize data

For LBP :

```
IMG_SIZE_LBP=(256,256)
def preprocess_image_LBP(image, label):
    """
    Resize and normalize the image.
    Args:
        image: The input image.
        label: The label corresponding to the image.
    Returns:
        Resized and normalized image, label.
    """
    # Resize the image
    image = tf.image.resize(image, IMG_SIZE_LBP)
    # Normalize the image to the range [0, 1]
    image = image / 255.0
    return image, label
```

What this code does?

Same but different size as it increases accuracy as more features is given

For Both :

```
train_dataset_original_hog = train_dataset_original_hog.map(preprocess_image_hog).batch(32).shuffle(1000)
test_dataset_original_hog = test_dataset_original_hog.map(preprocess_image_hog).batch(32)
```

For every image in the dataset, we preprocess it then divide them into batches each batch contains 32 images then we shuffle after 1000, which means that we change the order of images each 1000 images, so model does not memorize their places.

Visualize Data

```
[ ]  class_names = info.features['label'].names

    def visualize_data(dataset, num_samples=5):
        """
        Visualizes the first few images from the dataset.
        Args:
            dataset: The TensorFlow dataset.
            num_samples: Number of samples to display.
        """
        plt.figure(figsize=(10, 5))
        for i, (image, label) in enumerate(dataset.take(num_samples)):
            class_name = class_names[label.numpy()]
            plt.subplot(1, num_samples, i + 1)# 1 row 5 coulmns where to put image
            plt.imshow(image)
            plt.title(f"Label: {label.numpy()} - {class_name}")
            plt.axis("off")
        plt.subplots_adjust(wspace=4)  # Increase the space between images
        plt.show()

    # Visualize first 5 samples from the training set
    for images, labels in train_dataset_original_hog.take(1):
        visualize_data(tf.data.Dataset.from_tensor_slices((images, labels)))
```

We visualized 5 samples of a random batch.

HOG Feature Extraction

1) **For the extraction process itself**

```python
def extract_hog_features(dataset):
    hog_features = []
    labels = []

    # Iterate through the dataset (Note: dataset is batched)
    for images, batch_labels in dataset:
        # Convert the TensorFlow tensor to a NumPy array
        images_np = images.numpy()

        # Loop over each image in the batch and compute HOG features
        for img_np, label in zip(images_np, batch_labels.numpy()):
            # Convert the image to grayscale
            gray_img = rgb2gray(img_np)

            # Compute HOG features
            features = hog(gray_img,
                           pixels_per_cell=(8, 8),
                           cells_per_block=(2, 2),
                           block_norm='L1',
                           visualize=False)

            # Append the features and labels to the lists
            hog_features.append(features)
            labels.append(label)
```

We have two lists, one for label and one for hog features for each image.

We convert each image to gray then apply hog function then add the features extracted to the list

2) **Standardize Data**

```python
def standardize_data(train_features, test_features):
    scaler = StandardScaler()
    train_features = scaler.fit_transform(train_features)
    test_features = scaler.transform(test_features)
    return train_features, test_features
```

We standardize the data so we can

- Ensures all features contribute equally by removing the impact of their scales.
- Helps algorithms like logistic the support vector machine.

Features might have different scales:

- **Example:** Imagine we're describing a dog with these features:
    - **Tail length**: 20 cm
    - **Fur color (shade)**: 150 (grayscale intensity)
    - **Tongue length**: 10 cm

These features have different scales, so the model might focus more on the feature with larger values (e.g., fur color) and ignore the others. Standardization ensures that **all features contribute equally** by converting them into the same scale.

Key Equation for Standardization

$$X_{\text{standardized}} = \frac{X - \mu}{\sigma}$$

Where :

- X **:** Original Value
- μ **:** Mean of the feature
- σ **:** Standard deviation of the feature

Example of these calculations:

Train Features: [[20, 150, 10], [25, 200, 15], [30, 250, 20]]

Test Features: [[22, 180, 12], [28, 230, 18]]

- Columns represent:
    1. **Tail length** (cm)
    2. **Fur color (shade)**
    3. **Tongue length** (cm)

We will use the function to standardize these features step-by-step:

Step (1) : Calculate the following parameters:

Means (μ) :

$$\text{Tail length: } \mu = \frac{20+25+30}{3} = 25$$

$$\text{Fur color: } \mu = \frac{150+200+250}{3} = 200$$

$$\text{Tongue length: } \mu = \frac{10+15+20}{3} = 15$$

Standard Deviation (σ) :

- Tail length:

$$\sigma = \sqrt{\frac{(20-25)^2 + (25-25)^2 + (30-25)^2}{3}} = \sqrt{\frac{25}{3}} \approx 2.89$$

- Fur color:

$$\sigma = \sqrt{\frac{(150-200)^2 + (200-200)^2 + (250-200)^2}{3}} = \sqrt{\frac{5000}{3}} \approx 40.82$$

- Tongue length:

$$\sigma = \sqrt{\frac{(10-15)^2 + (15-15)^2 + (20-15)^2}{3}} = \sqrt{\frac{25}{3}} \approx 2.89$$

Step (2) :Standardize the Training Data

For **Tail Length** (column 1):

- $(20-25)/2.89 \approx -1.73$
- $(25-25)/2.89 = 0$
- $(30-25)/2.89 \approx 1.73$

For **Fur Color** (column 2):

- $(150-200)/40.82 \approx -1.22$
- $(200-200)/40.82 = 0$
- $(250-200)/40.82 \approx 1.22$

For **Tongue Length** (column 3):

- $(10-15)/2.89 \approx -1.73$
- $(15-15)/2.89 = 0$
- $(20-15)/2.89 \approx 1.73$

Standardized Training Data:

[[-1.73, -1.22, -1.73],

[0.00, 0.00, 0.00],

[1.73, 1.22, 1.73]]

Step (3) :

Use the same means (μ) and standard deviations (σ) from the training data:

- For **Tail Length**:
  - $(22 - 25)/2.89 \approx -1.04$
  - $(28 - 25)/2.89 \approx 1.04$
- For **Fur Color**:
  - $(180 - 200)/40.82 \approx -0.49$
  - $(230 - 200)/40.82 \approx 0.73$
- For **Tongue Length**:
  - $(12 - 15)/2.89 \approx -1.04$
  - $(18 - 15)/2.89 \approx 1.04$

Standardized Testing Data:

[[-1.04, -0.49, -1.04], [1.04, 0.73, 1.04]]

3) Apply Dimensions Reduction method (PCA)

```
def apply_pca(train_features, test_features, n_components=50):
    pca = PCA(n_components=n_components)
    train_features_pca = pca.fit_transform(train_features)
    test_features_pca = pca.transform(test_features)
    return train_features_pca, test_features_pca
```

fit_transform does two things:

- **Fit**: Computes the principal components using train_features. This involves:
  - Finding directions in the dataset with maximum variance.
  - These directions become the new "axes" for the data.
- **Transform**: Projects the training data onto these new axes, reducing its dimensions to n_components.

Step (1) : Take the standardized matrix

$$\text{Train (standardized)} \approx \begin{bmatrix} 0.78 & 0.43 & -0.8 & 0.76 \\ -1.26 & -1.38 & 1.2 & -1.08 \\ 0.48 & 0.95 & -0.4 & 0.35 \end{bmatrix}$$

Step (2) : Compute the Covariance matrix

The covariance matrix captures the relationships between features:

$$\text{Covariance Matrix} = \frac{1}{n-1} X_{\text{standardized}}^{\top} X_{\text{standardized}}$$

For our example, this matrix will be 4×4 (since there are 4 original features).

Step (3) and (4) : Eigenvalues and Eigenvectors

Perform eigen decomposition of the covariance matrix.

- **Eigenvalues**:

$$\text{Eigenvalues} = [2.72, 1.01, 0.23, 0.04]$$

- **Eigenvectors** (corresponding to the eigenvalues):

$$\text{Eigenvectors} = \begin{bmatrix} 0.52 & 0.26 & -0.48 & 0.65 \\ 0.51 & 0.27 & -0.53 & -0.62 \\ -0.47 & 0.68 & 0.38 & -0.42 \\ 0.51 & -0.61 & 0.57 & 0.27 \end{bmatrix}$$

↓

Step (5) : Reduce Dimensionality

Select the top 2 eigenvectors corresponding to the 2 largest eigenvalues:

$$\mathbf{P} = \begin{bmatrix} 0.52 & 0.26 \\ 0.51 & 0.27 \\ -0.47 & 0.68 \\ 0.51 & -0.61 \end{bmatrix}$$

Results :

For Training Data:

1. First Row:

$$[0.78, 0.43, -0.8, 0.76] \cdot \begin{bmatrix} 0.52 & 0.26 \\ 0.51 & 0.27 \\ -0.47 & 0.68 \\ 0.51 & -0.61 \end{bmatrix} \approx [1.8, 0.5]$$

2. Second Row:

$$[-1.26, -1.38, 1.2, -1.08] \cdot \begin{bmatrix} 0.52 & 0.26 \\ 0.51 & 0.27 \\ -0.47 & 0.68 \\ 0.51 & -0.61 \end{bmatrix} \approx [-0.7, -0.8]$$

3. Third Row:

4) Functions Calls and Implementation for HOG

```
# Extract HOG features for training and testing datasets

train_hog_features, train_labels_hog_features = extract_hog_features(train_dataset_original_hog)
test_hog_features, test_labels_hog_features = extract_hog_features(test_dataset_original_hog)

# Standardize the features
train_hog_features, test_hog_features = standardize_data(train_hog_features, test_hog_features)

# Apply PCA for dimensionality reduction
train_hog_features_pca, test_hog_features_pca = apply_pca(train_hog_features, test_hog_features, n_components=50)

# Train-test split for cross-validation
X_train_spilted_hog, X_valid_splited_hog, y_train_splited_hog, y_valid_splited_hog = train_test_split(
    train_hog_features_pca, train_labels_hog_features, test_size=0.2, random_state=42 #same randomization each run
)

# Verify the shape of HOG features
print(f"Train HOG features shape: {train_hog_features.shape}")
print(f"Test HOG features shape: {test_hog_features.shape}")

X_train_spilted_hog_p1 = np.copy(X_train_spilted_hog)
X_valid_splited_hog_p1 = np.copy(X_valid_splited_hog)
y_train_splited_hog_p1 = np.copy(y_train_splited_hog)
y_valid_splited_hog_p1 = np.copy(y_valid_splited_hog)

X_train_spilted_hog_p2 = np.copy(X_train_spilted_hog)
X_valid_splited_hog_p2 = np.copy(X_valid_splited_hog)
y_train_splited_hog_p2 = np.copy(y_train_splited_hog)
y_valid_splited_hog_p2 = np.copy(y_valid_splited_hog)
```

Local Binary Patterns Feature Extraction

```python
def extract_lbp_features(image, radius=1, n_points=8):
    """
    Extract Local Binary Pattern (LBP) features from an image.
    Args:
        image: Input image (H, W, C).
        radius: Radius of the LBP pattern.
        n_points: Number of points in the LBP pattern.
    Returns:
        A flattened LBP histogram.
    """
    # Convert image to grayscale
    gray_image = tf.image.rgb_to_grayscale(image).numpy().squeeze()

    # Compute the LBP
    lbp = local_binary_pattern(gray_image, n_points, radius, method="uniform")

    # Compute histogram
    hist, _ = np.histogram(
        lbp.ravel(),
        bins=np.arange(0, n_points + 3),  # Uniform LBP bins
        range=(0, n_points + 2),
    )
    hist = hist.astype("float")
    hist /= hist.sum()  # Normalize the histogram
    return hist

def preprocess_dataset(dataset):
    """
    Extract LBP features and labels from the dataset.
    """
    X = []
    y = []
    for image, label in dataset.unbatch():
        #lbp_features=extract_lbp_features(image, radius=2, n_points=16)
        hist1 = extract_lbp_features(image, radius=1, n_points=8)
        hist2 = extract_lbp_features(image, radius=2, n_points=16)
        hist3 = extract_lbp_features(image, radius=3, n_points=24)
        hist4 = extract_lbp_features(image, radius=4, n_points=48)
        hist5 = extract_lbp_features(image, radius=5, n_points=96)
        lbp_features = np.concatenate([hist1, hist2, hist3,hist4,hist5])
        X.append(lbp_features)
        y.append(label.numpy())

    # Plot the concatenated histogram
    return np.array(X), np.array(y)
```

```python
# Extract features
train_dataset_original_lbp = train_dataset_original_lbp.map(preprocess_image_LBP).batch(32).shuffle(1000)
test_dataset_original_lbp = test_dataset_original_lbp.map(preprocess_image_LBP).batch(32)

train_features_LBP, train_labels_LBP = preprocess_dataset(train_dataset_original_lbp)
test_features_LBP, test_labels_LBP = preprocess_dataset(test_dataset_original_lbp)

# Standardize the features
train_features_LBP, test_features_LBP = standardize_data(train_features_LBP, test_features_LBP)

# Apply PCA for dimensionality reduction
#train_features_LBP_pca, test_features_LBP_pca = apply_pca(train_features_LBP, test_features_LBP, n_components=50)

# Train-test split for cross-validation
X_train_spilted_LBP, X_valid_splited_LBP, y_train_splited_LBP, y_valid_splited_LBP = train_test_split(
    train_features_LBP, train_labels_LBP, test_size=0.2, random_state=42
)
print(f"Train LBP features shape: {train_features_LBP.shape}")
print(f"Test LBP features shape: {test_features_LBP.shape}")

X_train_spilted_LBP_p1 = np.copy(X_train_spilted_LBP)
X_valid_splited_LBP_p1 = np.copy(X_valid_splited_LBP)
y_train_splited_LBP_p1 = np.copy(y_train_splited_LBP)
y_valid_splited_LBP_p1 = np.copy(y_valid_splited_LBP)
```

Code Explanation (1) with example

```python
# Compute the LBP
lbp = local_binary_pattern(gray_image, n_points, radius, method="uniform")
```

Input:

- Grayscale values around the center pixel: `[60, 80, 90, 100, 70, 50, 40, 30]`
- Center pixel intensity: `70`
- Parameters: `n_points=8`, `radius=1`, `method="uniform"`

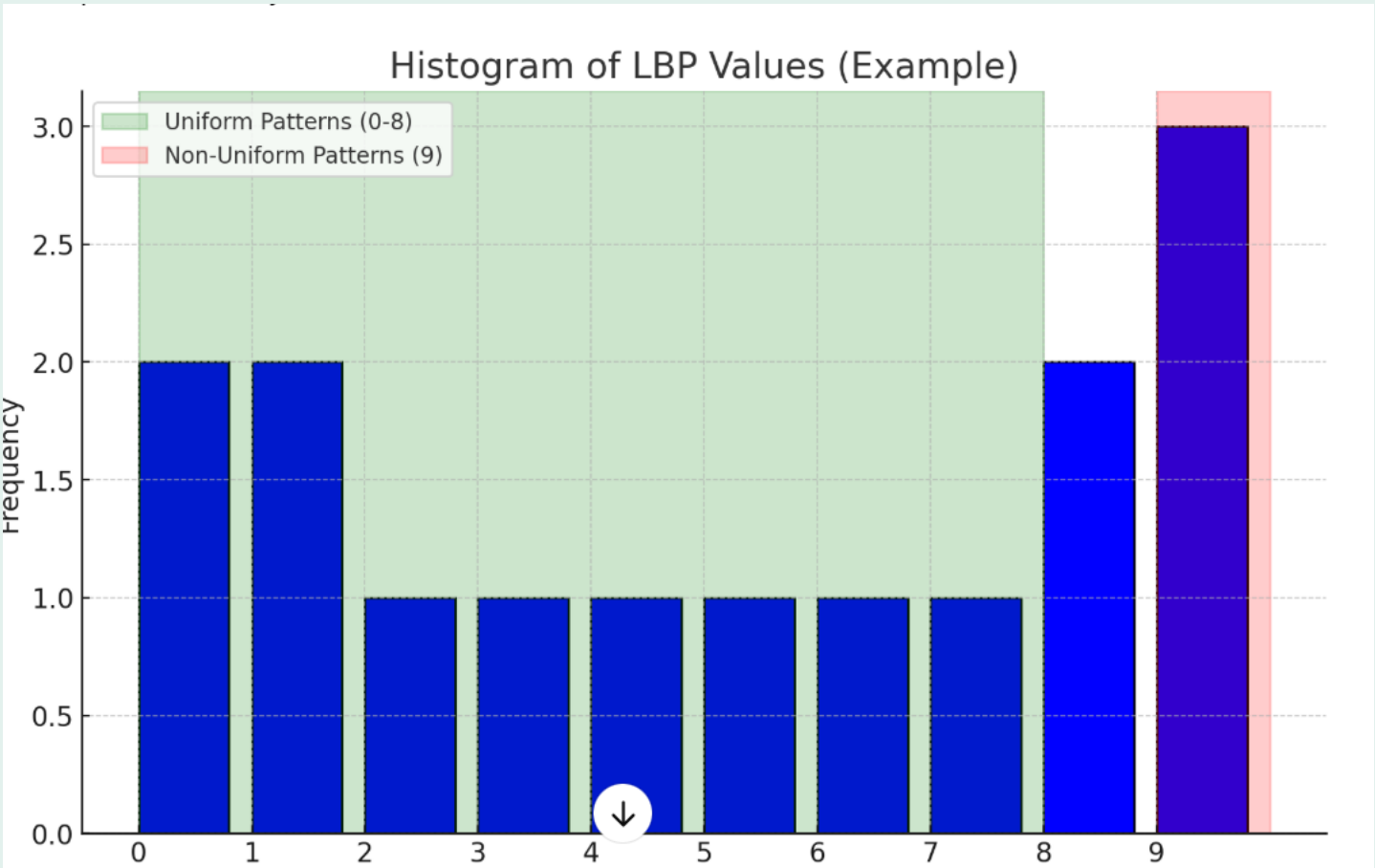
Process:

1. Compare neighbors to the center pixel:

$$\text{Binary pattern} = [0, 1, 1, 1, 0, 0, 0, 0]$$

2. Circular sequence: `01110000`.
3. Check uniformity:
   - Transitions: $0 \rightarrow 1 \rightarrow 0$ (2 transitions, **uniform**).
4. Convert to decimal:
   - $01110000 \rightarrow 112$.

Visual representation of the output :



Code Explanation (2) with example

```
# Convert the image to HSV color space
image_hsv = tf.image.rgb_to_hsv(image)
```

What is HSV and RGB?

- **RGB (Red, Green, Blue)**: This color space is based on **light intensities** in the red, green, and blue channels. It's commonly used in digital displays and imaging but doesn't always align with how humans perceive color.
- **HSV (Hue, Saturation, Value)**: This color space is often more **intuitive** for humans to understand and use. Here's what the components represent:
    - **Hue (H)**: Represents the **type of color** (e.g., red, blue, yellow). It's the **color angle** in the color wheel.
    - **Saturation (S)**: Represents how **intense** or **pure** the color is. High saturation means the color is vibrant, while low saturation means the color is more washed out (closer to gray).
    - **Value (V)**: Represents the **brightness** or **lightness** of the color. Higher values mean the color is lighter, and lower values mean it's darker.

Example: you're working with an image of a **red dog** in different lighting conditions:

1. In **RGB**, the exact values of red may vary due to changes in lighting:
    - In **low light**, the red component of the dog might be lower (e.g., RGB = [50, 0, 0]).
    - In **bright light**, the red component might be higher (e.g., RGB = [200, 0, 0]).

    But in **HSV**, the **Hue** for both cases could remain similar (because both are red), while the **Saturation** and **Value** change according to the lighting. This makes it easier to track the color of the dog (red) across different lighting conditions, without being affected by changes in brightness.

2. When extracting a **color histogram** from an **HSV image**, we're mainly interested in:
    - **Hue**: Captures the **color** (red, blue, green, etc.).
    - **Saturation and Value**: Can provide insights into **intensity** and **brightness** but are less important for distinguishing colors.

    By focusing on the **Hue** component, you can build a more **robust histogram** that captures the main colors of the image, regardless of changes in light.

Color Histogram Feature Extraction

```python
IMG_SIZE_COLOR_HIST = (256, 256)

def preprocess_image_color_hist(image, label):
    """
    Resize the image and convert it to HSV for color histogram extraction.
    Args:
        image: The input image.
        label: The label corresponding to the image.
    Returns:
        Resized HSV image, label.
    """
    # Resize the image
    image = tf.image.resize(image, IMG_SIZE_COLOR_HIST)

    # Convert the image to HSV color space
    image_hsv = tf.image.rgb_to_hsv(image)

    return image_hsv, label

def extract_color_histogram(image, bins=32):
    """
    Extract a color histogram from an image.
    Args:
        image: Input image (H, W, C) in RGB format.
        bins: Number of bins for the histogram.
    Returns:
        A concatenated histogram for R, G, and B channels.
    """
    # Convert the TensorFlow tensor to a NumPy array
    image_np = image.numpy()

    # Compute histograms for each channel (R, G, B)
    hist_r, _ = np.histogram(image_np[:, :, 0], bins=bins, range=(0, 255))
    hist_g, _ = np.histogram(image_np[:, :, 1], bins=bins, range=(0, 255))
    hist_b, _ = np.histogram(image_np[:, :, 2], bins=bins, range=(0, 255))

    # Normalize histograms
    hist_r = hist_r / hist_r.sum()
    hist_g = hist_g / hist_g.sum()
    hist_b = hist_b / hist_b.sum()

    # Concatenate histograms into a single feature vector
    color_histogram = np.concatenate([hist_r, hist_g, hist_b])
    return color_histogram
```

```python
def extract_color_histogram_with_spatial(image, bins=128, grid_size=8):
    """
    Extract color histograms from spatially partitioned regions of an image.
    Args:
        image: Input image (H, W, C).
        bins: Number of bins for the histogram.
        grid_size: Number of divisions along each axis (e.g., 4x4 grid).
    Returns:
        Concatenated histograms from all grid regions.
    """
    h, w, c = image.shape
    grid_h, grid_w = h // grid_size, w // grid_size
    histograms = []
    for i in range(grid_size):
        for j in range(grid_size):
            # Extract region
            region = image[i * grid_h:(i + 1) * grid_h, j * grid_w:(j + 1) * grid_w]
            # Extract histogram for the region
            histograms.append(extract_color_histogram(region, bins))
    return np.concatenate(histograms)

def preprocess_dataset_with_color(dataset, bins=32):
    """
    Extract color histogram features and labels from the dataset.
    Args:
        dataset: The input TensorFlow dataset.
        bins: Number of bins for the color histogram.
    Returns:
        Feature matrix (X) and labels (y).
    """
    X = []
    y = []

    for element in dataset:
        # If the dataset contains (image, label) tuples
        if isinstance(element, tuple):
            image, label = element
        else:
            # Adjust if the dataset format is scalar or needs unpacking differently
            raise ValueError("Unexpected dataset structure: ensure it's in (image, label) format.")

        # Extract color histogram features
        color_histogram = extract_color_histogram_with_spatial(image, bins=128)

        # Combine feature
        X.append(color_histogram)
        y.append(label.numpy())

    return np.array(X), np.array(y)
```

```
train_dataset_original_color = train_dataset_original_color.map(preprocess_image_color_hist).batch(32).shuffle(1000).unbatch()
test_dataset_original_color = test_dataset_original_color.map(preprocess_image_color_hist).batch(32).unbatch()

# Extract color histogram features
train_features_color, train_labels_color = preprocess_dataset_with_color(train_dataset_original_color, bins=128)
test_features_color, test_labels_color = preprocess_dataset_with_color(test_dataset_original_color, bins=128)


# Standardize the features
train_features_color, test_features_color = standardize_data(train_features_color, test_features_color)

# Apply PCA for dimensionality reduction
train_features_color_pca, test_features_color_pca = apply_pca(train_features_color, test_features_color, n_components=100)

# Train-test split for cross-validation
X_train_split_color, X_valid_split_color, y_train_split_color, y_valid_split_color = train_test_split(
    train_features_color_pca, train_labels_color, test_size=0.2, random_state=42
)

print(f"Train Color Histogram features shape: {train_features_color.shape}")
print(f"Test Color Histogram features shape: {test_features_color.shape}")

# Example: Split the training data for validation
X_train_spilted_color_p1 = np.copy(X_train_split_color)
X_valid_splited_color_p1 = np.copy(X_valid_split_color)
y_train_splited_color_p1 = np.copy(y_train_split_color)
y_valid_splited_color_p1 = np.copy(y_valid_split_color)
```

KNN Method

```python
def euclidean_distance(x1, x2):
    """
    Compute the Euclidean distance between vectors.
    Args:
        x1: Single vector or multiple vectors (1D or 2D).
        x2: Single vector (1D).
    Returns:
        Distance(s) as a scalar or array.
    """
    x1 = np.array(x1)
    x2 = np.array(x2)
    return np.sqrt(np.sum((x1 - x2) ** 2, axis=-1))  # Handles 1D and 2D

def knn_predict(train_data, train_labels, test_data, k):
    """
    Predict the label for a test data point using the k-nearest neighbors algorithm.

    Args:
    - train_data (np.ndarray): Training feature vectors.
    - train_labels (list or np.ndarray): Corresponding labels for the training data.
    - test_data (np.ndarray): A single test feature vector.
    - k (int): Number of neighbors to consider.

    Returns:
    - predicted_label: The label predicted for the test data.
    """
    # Compute distances and pair each with the corresponding label
    distances_and_labels = [
        (euclidean_distance(test_data, x_train), train_labels[i])
        for i, x_train in enumerate(train_data)
    ]

    # Sort pairs by distance
    sorted_neighbors = sorted(distances_and_labels, key=lambda x: x[0])

    # Extract the labels of the k nearest neighbors
    k_labels = [label for _, label in sorted_neighbors[:k]]

    # Majority vote to determine the predicted label
    predicted_label = Counter(k_labels).most_common(1)[0][0]

    return predicted_label
```

SVM Method

```
#Train SVM classifier
"""
scaler_hog = StandardScaler()
X_train_hog_SVM = scaler_hog.fit_transform(X_train_spilted_hog)
X_valid_hog_SVM = scaler_hog.transform(X_valid_splited_hog)
X_test_hog_SVM = scaler_hog.transform(test_hog_features)
"""
# Train SVM classifier
svm_classifier_HOG = svm.SVC(C=10,kernel='rbf')  # You can also try 'rbf' kernel for better accuracy
svm_classifier_HOG.fit(X_train_spilted_hog, y_train_splited_hog)

# Make predictions on the test set
y_pred_hog = svm_classifier_HOG.predict(test_hog_features_pca)

# Evaluate accuracy
accuracy = accuracy_score(test_labels_hog_features, y_pred_hog)
print(f'Accuracy on test (HOG) set: {accuracy * 100:.2f}%')


#Train SVM classifier
"""
scaler_LBP = StandardScaler()
X_train_LBP_SVM = scaler_LBP.fit_transform(X_train_spilted_LBP)
X_valid_LBP_SVM = scaler_LBP.transform(X_valid_splited_LBP) test_features_LBP_pca
X_test_LBP_SVM = scaler_LBP.transform(test_features_LBP)
"""
# Train SVM classifier
svm_classifier_lbp = svm.SVC(C=10,kernel='rbf') # You can also try 'rbf' kernel for better accuracy
svm_classifier_lbp.fit(X_train_spilted_LBP, y_train_splited_LBP)

# Make predictions on the test set
y_pred_LBP = svm_classifier_lbp.predict(test_features_LBP)

# Evaluate accuracy
accuracy = accuracy_score(test_labels_LBP, y_pred_LBP)
print(f'Accuracy on test (LBP) set: {accuracy * 100:.2f}%')

#Train SVM classifier
"""
scaler_color = StandardScaler()
X_train_color_SVM = scaler_color.fit_transform(X_train_split_color)
X_valid_color_SVM = scaler_color.transform(X_valid_split_color)
X_test_color_SVM = scaler_color.transform(test_features_color)
"""
# Train SVM classifier
svm_classifier_color = svm.SVC(C=10,kernel='rbf')  # You can also try 'rbf' kernel for better accuracy
svm_classifier_color.fit(X_train_split_color, y_train_split_color)

# Make predictions on the test set
y_pred_color = svm_classifier_color.predict(test_features_color_pca)

# Evaluate accuracy
accuracy = accuracy_score(test_labels_color, y_pred_color)
print(f'Accuracy on test (color histogram) set: {accuracy * 100:.2f}%')
```

K-Means Method

```
# Fit the K-Means model to the training data (using HOG features)
k = 102  # Number of clusters for Caltech101 dataset (102 classes)
print("HOG RESULTS")
y_train_clusters = K_Means_fit(X_train_spilted_hog_p2, k=k, max_iterations=2000)

y_train_pred = map_clusters_to_labels(y_train_splited_hog_p2, y_train_clusters, k)

# Calculate accuracy
train_accuracy = accuracy_score(y_train_splited_hog_p2, y_train_pred)
print(f'Clustering accuracy on training set: {train_accuracy * 100:.2f}%')
# Predict clusters for the validation set using the centroids obtained from training
y_valid_clusters = K_Means_fit(X_valid_splited_hog_p2, k=k, max_iterations=200000)

# Map clusters to labels for the validation set
y_valid_pred = map_clusters_to_labels(y_valid_splited_hog_p2, y_valid_clusters, k)

# Calculate accuracy for the validation set
valid_accuracy = accuracy_score(y_valid_splited_hog_p2, y_valid_pred)
print(f'Clustering accuracy on validation set: {valid_accuracy * 100:.2f}%')
################################################################################
print("LBP RESULTS")
y_train_clusters = K_Means_fit(X_train_spilted_LBP, k=k, max_iterations=2000)

y_train_pred = map_clusters_to_labels(y_train_splited_LBP, y_train_clusters, k)

# Calculate accuracy
train_accuracy = accuracy_score(y_train_splited_LBP, y_train_pred)
print(f'Clustering accuracy on training set: {train_accuracy * 100:.2f}%')
# Predict clusters for the validation set using the centroids obtained from training
y_valid_clusters = K_Means_fit(X_valid_splited_LBP, k=k, max_iterations=200000)

# Map clusters to labels for the validation set
y_valid_pred = map_clusters_to_labels(y_valid_splited_LBP, y_valid_clusters, k)

# Calculate accuracy for the validation set
valid_accuracy = accuracy_score(y_valid_splited_LBP, y_valid_pred)
print(f'Clustering accuracy on validation set: {valid_accuracy * 100:.2f}%')
################################################################################
print("COLOR RESULTS")
y_train_clusters = K_Means_fit(X_train_spilted_color_p1, k=k, max_iterations=2000)

y_train_pred = map_clusters_to_labels(y_train_splited_color_p1, y_train_clusters, k)

# Calculate accuracy
train_accuracy = accuracy_score(y_train_splited_color_p1, y_train_pred)
print(f'Clustering accuracy on training set: {train_accuracy * 100:.2f}%')
# Predict clusters for the validation set using the centroids obtained from training
y_valid_clusters = K_Means_fit(X_valid_splited_color_p1, k=k, max_iterations=200000)

# Map clusters to labels for the validation set
y_valid_pred = map_clusters_to_labels(y_valid_splited_color_p1, y_valid_clusters, k)

# Calculate accuracy for the validation set
valid_accuracy = accuracy_score(y_valid_splited_color_p1, y_valid_pred)
print(f'Clustering accuracy on validation set: {valid_accuracy * 100:.2f}%')
```

1) **What does this function?**

Purpose: Implements the K-Means clustering algorithm.

```
def K_Means_fit(X,k=3, max_iterations=200):
    global centroids
    centroids=np.random.uniform(np.amin(X,axis=0),np.amax(X,axis=0) ,size=(k,X.shape[1]))
    for _ in range(max_iterations):
        y=[]
        for data_point in X:
            distances=euclidean_distance(data_point,centroids)
            cluster_num=np.argmin(distances)
            y.append(cluster_num)
        y=np.array(y)
        cluster_index=[]

        for i in range(k):
            cluster_index.append(np.argwhere(y==i))

        cluster_centers=[]

        for i,index in enumerate(cluster_index):
            if len(index)==0:
                cluster_centers.append(centroids[i])
            else:
                cluster_centers.append(np.mean(X[index],axis=0)[0])

        if np.max(centroids-np.array(cluster_centers))<0.001:
            break
        else:
            centroids=np.array(cluster_centers)
    return y
```

Parameters:

- X : Dataset, a 2D array where each row is a data point.
- k: Number of clusters (102).
- max_iterations: Max number of iterations for the algorithm.

Step (1) : Randomly initializes k centroids within the range of the dataset X.
Example:

- If X has values between [0, 10], centroids will be within that range.
- For k=2 and X=[[1, 2], [3, 4], [5, 6]]: Centroids could start as [[2.1, 3.5], [4.7, 5.9]].

Step (2) : For each data point:

1. Calculate distances to all centroids.
2. Assign it to the closest centroid (cluster_num).

Example:

- If distances to centroids [2.1, 3.5] and [4.7, 5.9] are [1.2, 0.9], assign to the second centroid.
- Groups indices of points belonging to each cluster.
    - Example: If y = [0, 1, 1, 0] for 4 points and k=2, cluster_index becomes [[[0], [3]], [[1], [2]]].
- Calculates the new centroid as the mean of points in each cluster.
- If a cluster is empty, keep the old centroid.
    - Example: For points [1, 2] and [3, 4] in cluster 0, the new centroid is [2, 3].
- Stops if centroids no longer change significantly (convergence).

2) **What does this function?**

Purpose: Assign for each cluster a suitable label

```python
def map_clusters_to_labels(y_true, y_clusters, k):
    cluster_to_label = {}

    # For each cluster, find the most common label in that cluster
    for cluster in range(k):
        cluster_indices = np.where(y_clusters == cluster)[0]

        # Check if the cluster has data points
        if len(cluster_indices) > 0:
            cluster_labels = y_true[cluster_indices]

            # Majority voting to assign the most common label to the cluster
            most_common_label = np.bincount(cluster_labels).argmax()
            cluster_to_label[cluster] = most_common_label
        else:
            # If the cluster is empty, you can assign a default label (e.g., -1 or any label of your choice)
            cluster_to_label[cluster] = -1  # Or choose any default label

    # Map predicted clusters to actual labels
    y_pred_mapped = np.array([cluster_to_label.get(cluster, -1) for cluster in y_clusters])
    return y_pred_mapped
```

Parameters:

- y_clusters: Array where each value is the cluster number for a data point. Example: [0, 1, 1, 0, 2] means the 1st and 4th points belong to cluster 0, the 2nd and 3rd to cluster 1, and the 5th to cluster 2.
- y_true: Array of actual labels for the data points. Example: [2, 3, 3, 2, 4].
- k is the number of clusters.

Step (1) with an example :

- y_true = [0, 1, 1, 0], y_clusters = [0, 1, 1, 0].
- Cluster 0 has labels [0, 0], so assign label 0.
- Cluster 1 has labels [1, 1], so assign label 1.

Step (2) with an example : If k=3, this loop runs for cluster = 0, 1, 2.

- Finds the indices of all data points assigned to the current cluster.

Step (3) with an example :

If y_clusters = [0, 1, 1, 0, 2] and cluster = 1, then cluster_indices = [1, 2].

- If the cluster is empty (len(cluster_indices) == 0), skip to the next cluster.
- Example: If y_clusters = [0, 1, 1, 0, 2] and cluster = 3, this condition is false because no points belong to cluster 3.
- Retrieves the labels of points in the current cluster using y_true.

Step (4) with an example : Majority Voting

- If y_true = [2, 3, 3, 2, 4] and cluster_indices = [1, 2] (from step 2),
  - Then cluster_labels = [3, 3].
  - **np.bincount(cluster_labels)**: Counts how many times each label appears.
    Example: For cluster_labels = [3, 3], the result is [0, 0, 0, 2].
    Index 0 has 0 occurrences.
    Index 1 has 0 occurrences.
    Index 2 has 0 occurrences.
    Index 3 has 2 occurrences.
  - **.argmax()**: Finds the label with the highest count.
    Example: For [0, 0, 0, 2], the maximum value is 2, and its index is 3.

Step (5) with an example : Maps the current cluster to its most common label.

- For cluster = 1, most_common_label = 3.
- So, cluster_to_label[1] = 3.

Final Output Format :

- After the loop finishes, cluster_to_label is a dictionary where:
- Keys are cluster numbers.
- As values are the most common labels in those clusters.

This function returns

- y_clusters = [0, 1, 1, 0, 2]
- y_true = [2, 3, 3, 2, 4]

Example of the Final Output:

- cluster_to_label = {0: 2, 1: 3, 2: 4}

This means:

- Cluster 0 is mapped to label 2.
- Cluster 1 is mapped to label 3.
- Cluster 2 is mapped to label 4.
  - For a specific cluster (e.g., cluster 1), this gives the indices of all points assigned to it.

This means : If y_clusters = [0, 1, 1, 0, 2] and cluster = 1, cluster_indices = [1, 2].

- Use the indices to fetch the corresponding labels from y_true.

This means : If cluster_indices = [1, 2] and y_true = [2, 3, 3, 2, 4], then cluster_labels = [3, 3].

Step (6) with an example : Creates an array where the value at each index represents the count of that label.

Inputs:

- y_clusters = [0, 1, 1, 0, 2] (cluster assignments)
- y_true = [2, 3, 3, 2, 4] (true labels)
- cluster = 1 (current cluster we're analyzing)

Recap on steps (4) to (6):

- Cluster 1 (from y_clusters) contains the 2nd and 3rd data points.
- The true labels for those points (from y_true) are [3, 3].
- The most common label in cluster 1 is 3.
- This process is repeated for every cluster (cluster = 0, 1, 2, ...) to map clusters to the most common labels.

```
# Fit the K-Means model to the training data (using HOG features)
k = 102  # Number of clusters for Caltech101 dataset (102 classes)
print("HOG RESULTS")
y_train_clusters = K_Means_fit(X_train_spilted_hog_p2, k=k, max_iterations=2000)

y_train_pred = map_clusters_to_labels(y_train_splited_hog_p2, y_train_clusters, k)

# Calculate accuracy
train_accuracy = accuracy_score(y_train_splited_hog_p2, y_train_pred)
print(f'Clustering accuracy on training set: {train_accuracy * 100:.2f}%')
# Predict clusters for the validation set using the centroids obtained from training
y_valid_clusters = K_Means_fit(X_valid_splited_hog_p2, k=k, max_iterations=200000)

# Map clusters to labels for the validation set
y_valid_pred = map_clusters_to_labels(y_valid_splited_hog_p2, y_valid_clusters, k)

# Calculate accuracy for the validation set
valid_accuracy = accuracy_score(y_valid_splited_hog_p2, y_valid_pred)
print(f'Clustering accuracy on validation set: {valid_accuracy * 100:.2f}%')
################################################################################
print("LBP RESULTS")
y_train_clusters = K_Means_fit(X_train_spilted_LBP, k=k, max_iterations=2000)
y_train_pred = map_clusters_to_labels(y_train_splited_LBP, y_train_clusters, k)

# Calculate accuracy
train_accuracy = accuracy_score(y_train_splited_LBP, y_train_pred)
print(f'Clustering accuracy on training set: {train_accuracy * 100:.2f}%')

# Predict clusters for the validation set using the centroids obtained from training
y_valid_clusters = K_Means_fit(X_valid_splited_LBP, k=k, max_iterations=200000)

# Map clusters to labels for the validation set
y_valid_pred = map_clusters_to_labels(y_valid_splited_LBP, y_valid_clusters, k)

# Calculate accuracy for the validation set
valid_accuracy = accuracy_score(y_valid_splited_LBP, y_valid_pred)
print(f'Clustering accuracy on validation set: {valid_accuracy * 100:.2f}%')
################################################################################
print("COLOR RESULTS")
y_train_clusters = K_Means_fit(X_train_spilted_color_p1, k=k, max_iterations=2000)
y_train_pred = map_clusters_to_labels(y_train_splited_color_p1, y_train_clusters, k)

# Calculate accuracy
train_accuracy = accuracy_score(y_train_splited_color_p1, y_train_pred)
print(f'Clustering accuracy on training set: {train_accuracy * 100:.2f}%')
# Predict clusters for the validation set using the centroids obtained from training
y_valid_clusters = K_Means_fit(X_valid_splited_color_p1, k=k, max_iterations=200000)

# Map clusters to labels for the validation set
y_valid_pred = map_clusters_to_labels(y_valid_splited_color_p1, y_valid_clusters, k)

# Calculate accuracy for the validation set
valid_accuracy = accuracy_score(y_valid_splited_color_p1, y_valid_pred)
print(f'Clustering accuracy on validation set: {valid_accuracy * 100:.2f}%')
```

Google Drive Link

https://drive.google.com/drive/folders/1-MYEB4sGMGoSUzA8Uw-naY0nIxQJx7_s

GitHub Repository

git@github.com:MariamMedhat-gif/Major-Task_Machine-Vision.git

Colab Files

https://colab.research.google.com/drive/1dfyKwcPArKFOoC73F0mXcYwtzmZt49lt?usp=sharing#scrollTo=7dnj0I_2W3ag