Çağrı Çakıroğlu
2200765005
November,2023

AIN433_ASSIGNMENT 2

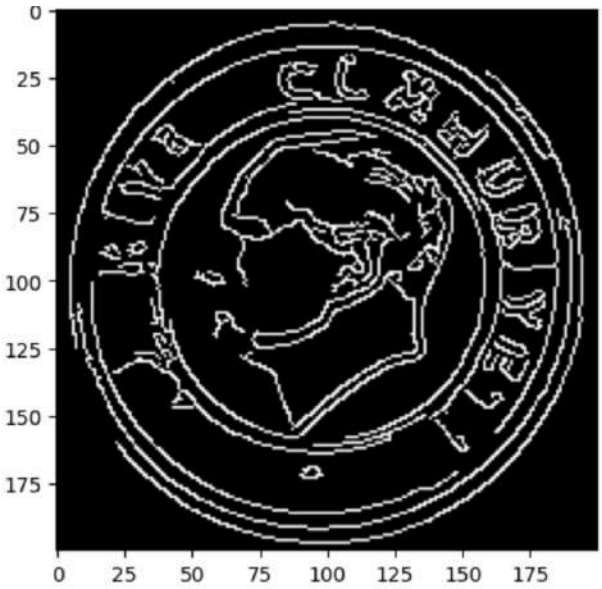
PART 1

Edge Detector Explanation and Parameter Setting

In this project, the Canny edge detection algorithm was chosen for identifying the edges in the image. This method was selected due to its effectiveness in reducing noise while accurately detecting edge boundaries. The Canny edge detector is known for its precision and is widely used in various image processing tasks.

The parameters of the Canny edge detection algorithm were carefully set to optimize edge detection. A Gaussian filter with a kernel size of 3×3 and a sigma value of 2 was applied, resulting in a smoothed image. This smoothing process helps in reducing noise and improving the clarity of the edges. The Canny function then used two threshold values, 30 and 220, for the hysteresis procedure. The lower threshold of 30 was set to identify the initial weak edges, while the higher threshold of 220 was employed to detect strong edges. This range was chosen to ensure that most relevant edges are captured without including too much noise.

Following the Canny edge detection, the Sobel operator was applied to compute the gradient direction for each pixel. This involved calculating the Sobel derivatives in the x and y directions using a kernel size of 5. The gradient direction at each pixel was then determined from these derivatives. This step is crucial for the subsequent Hough transform circle detection, as it provides the orientation information needed for accurately locating circles in the image.



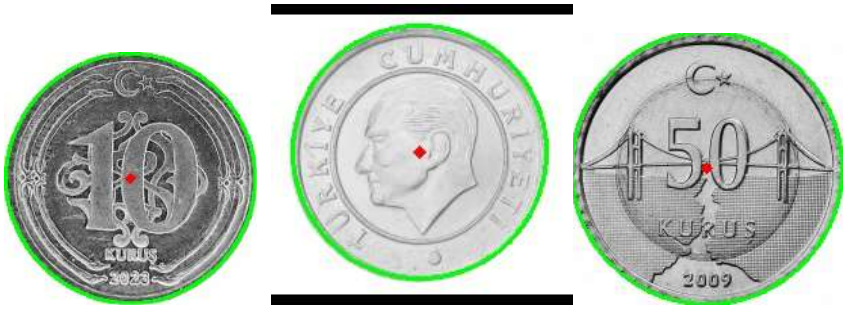
Canny Edge Detector Example on Train

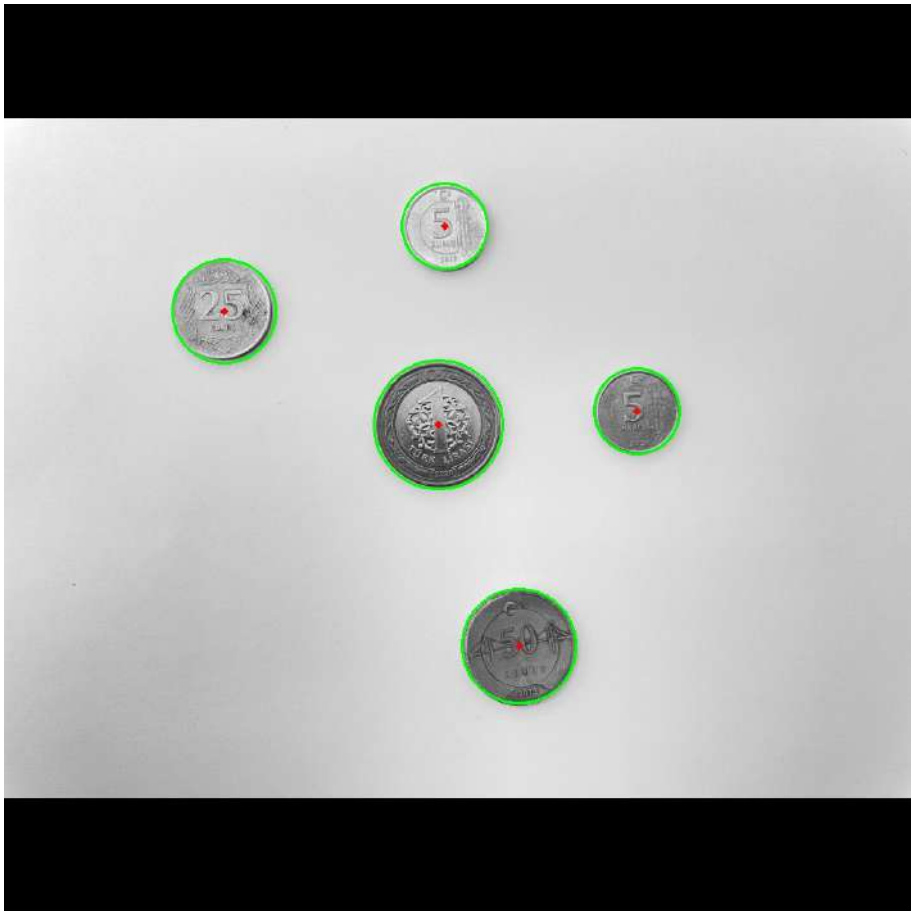
```python
def hough_transform_circle_detection(image, radius_range, threshold, output_path):
    # Step 1: Preprocessing
    global has_plotted_sobel


    diameter = 3  # Diameter of each pixel neighborhood
    sigmaColor = 100  # Filter sigma in the color space
    sigmaSpace = 100  # Filter sigma in the coordinate space
    filtered_image = cv2.bilateralFilter(image, diameter, sigmaColor, sigmaSpace)


    smoothed_image = cv2.GaussianBlur(filtered_image, (3, 3), 2)
    edges = cv2.Canny(smoothed_image, 30, 220)

    # Compute the gradient direction for each pixel
    sobelx = cv2.Sobel(smoothed_image, cv2.CV_64F, 1, 0, ksize=5)
    sobely = cv2.Sobel(smoothed_image, cv2.CV_64F, 0, 1, ksize=5)
    direction = np.arctan2(sobely, sobelx)
    magnitude = np.sqrt(sobelx**2 + sobely**2)
```





Few Examples of Circle Detection

Image Resizing Explanation

In this process, an image is resized to a specified size while maintaining its aspect ratio. The aspect ratio is the proportional relationship between the width and height of the image. Keeping this ratio constant ensures that the resized image does not appear stretched or squashed.

First, the dimensions of the original image are obtained. The width and height of the new size are also provided. The resizing process involves calculating the ratio of the new width or height to the original dimensions. If the original image is wider than it is tall, the width ratio is used to determine the new dimensions. Conversely, if the image is taller than it is wide, the height ratio is used.

After calculating the new dimensions, the image is resized using the cv2.resize function with an interpolation method. The INTER_AREA interpolation method is chosen for this operation, which is effective for resizing.

However, directly resizing the image to the new dimensions might not always match the desired size due to the aspect ratio maintenance. Therefore, additional steps are taken. If the resized image is smaller than the desired size, padding is added around the image. If it is larger, the image is cropped. This padding or cropping ensures that the final image matches the specified dimensions while retaining the correct aspect ratio.

The padding color is set to black, but this can be changed depending on the requirements of the application. The cv2.copyMakeBorder function is used to add the padding, resulting in the final resized image.

The main goal of this process is to ensure that the final image matches the desired size without distorting its original appearance.

```python
def resize_image(image, size):
    # Calculate the ratio of the new size and find the best match for the new dimensions
    # maintaining the aspect ratio.
    h, w = image.shape[:2]
    (width, height) = size

    # Calculate the ratio of the width and construct the dimensions
    if w > h:
        aspect = width / float(w)
        dim = (width, int(h * aspect))
    else:
        aspect = height / float(h)
        dim = (int(w * aspect), height)

    # Resize the image
    resized_image = cv2.resize(image, dim, interpolation=cv2.INTER_AREA)

    # If the new size is smaller, then we need to pad the image
    # If the new size is larger, we need to crop the image
    delta_w = width - resized_image.shape[1]
    delta_h = height - resized_image.shape[0]
    top, bottom = delta_h // 2, delta_h - (delta_h // 2)
    left, right = delta_w // 2, delta_w - (delta_w // 2)

    # Create a border around the image to maintain the aspect ratio
    color = [0, 0, 0]  # 'color' can be changed depending on the application
    new_resized_image = cv2.copyMakeBorder(resized_image, top, bottom, left, right, cv2.BORDER_CONSTANT, value=color)

    return new_resized_image
```

GrayScaling the resized images

```
gray_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)
```

Hough Transform Circle Detection Explanation

In this procedure, circles in an image are detected using the Hough Transform method. The process is initiated by preprocessing the image to enhance its features for better circle detection.

Initially, the image undergoes a bilateral filter. This filter is applied to reduce noise while preserving edges, using specific parameters like the diameter of each pixel neighborhood and filter sigma values in both color and coordinate spaces.

Next, the image is further smoothed using a Gaussian Blur. This step helps in reducing fine details and noise, making it easier to identify prominent edges. The parameters for the Gaussian Blur, such as the kernel size and sigma value, are carefully chosen.

After smoothing, edges in the image are detected using the Canny edge detector. This method is widely recognized for its ability to detect a wide range of edges in images. In the code, specific threshold values are set for the Canny function to identify significant edges.

Subsequently, the Sobel operator is used to compute the gradient direction for each pixel. This information is crucial for the Hough Transform as it helps in identifying potential circle centers and radii in the image. The Sobel operator calculates the derivative in both x and y directions, and from these, the magnitude and direction of the gradient are derived.

An important part of the process involves visualizing the detected edges. The code includes a condition to display the edges using the Sobel operator only once. This visualization is vital for understanding how effectively the edges have been detected.

The core of the circle detection lies in the Hough Transform. Here, a three-dimensional Hough space is initialized, corresponding to the x and y coordinates of the image and the range of possible radii for the circles. Votes are cast in this space for potential circle centers. This is done by iterating through each edge pixel and calculating where the center of a potential circle could be, based on the gradient direction and different radii.

After casting votes, local maxima in the Hough space are identified. These maxima correspond to the center coordinates and radii of the most likely circles. A threshold is set to determine significant peaks, indicating strong circle candidates.

The identified circles might have overlaps. A function is used to merge significantly overlapping circles, ensuring that each detected circle is distinct.

Finally, the detected circles are drawn onto the original image. This is done by outlining each circle and marking its center. The resulting image, showing the detected circles, is then saved to the specified output path.

This entire process demonstrates a robust method for circle detection in images, utilizing the Hough Transform technique combined with effective preprocessing and edge detection steps.

Explanation of Circle Merging and Overlap Calculation

In this process, overlapping circles detected in an image are merged to identify distinct circles accurately.

Firstly, circles that significantly overlap with each other are grouped together. To achieve this, each circle is compared with every other circle to determine if they overlap. The function calculate_circle_overlap is used for this purpose. In this function, the overlap between two circles is determined by calculating the distance between their centers and comparing it with the sum of their radii. If the distance is less than the sum of the radii, the circles are considered to be overlapping.

During this comparison, any circle that has already been grouped is skipped, ensuring that each circle is considered only once. As a result, groups of overlapping circles are formed. Each group contains either a single circle or multiple circles that overlap with each other.

After the groups are formed, the next step is to merge these groups into single circles. For groups with more than one circle, a new circle is created by averaging the x and y coordinates of the center points of all the circles in the group. The radius of the new circle is set to the largest radius found in the group. This approach ensures that the new circle encompasses all the circles in the group.

For groups with only one circle, no merging is needed, and the circle is included as is in the final list of circles.

The result of this process is a list of merged circles, where significantly overlapping circles are represented as single circles. This method helps in reducing redundancy and improves the accuracy of circle detection in the image.

```python
def calculate_circle_overlap(circle1, circle2):
    x1, y1, r1 = circle1
    x2, y2, r2 = circle2
    distance = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
    return distance < (r1 + r2)
```

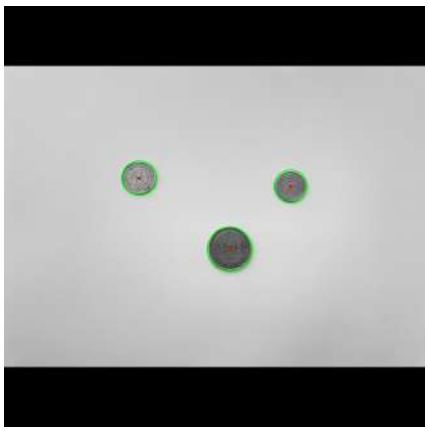
```python
def merge_significantly_overlapping_circles(circles):
    groups = []
    used = set()

    # Group overlapping circles
    for i, circle1 in enumerate(circles):
        if i in used:
            continue
        group = [circle1]
        for j, circle2 in enumerate(circles):
            if j in used or j == i:
                continue
            if calculate_circle_overlap(circle1, circle2):
                group.append(circle2)
                used.add(j)
        groups.append(group)

    # Merge groups into single circles
    merged_circles = []
    for group in groups:
        if len(group) > 1:
            avg_x = int(sum(x for x, _, _ in group) / len(group))
            avg_y = int(sum(y for _, y, _ in group) / len(group))
            max_r = max(r for _, _, r in group)
            merged_circles.append((avg_x, avg_y, max_r))
        else:
            merged_circles.extend(group)

    return merged_circles
```

More Examples:



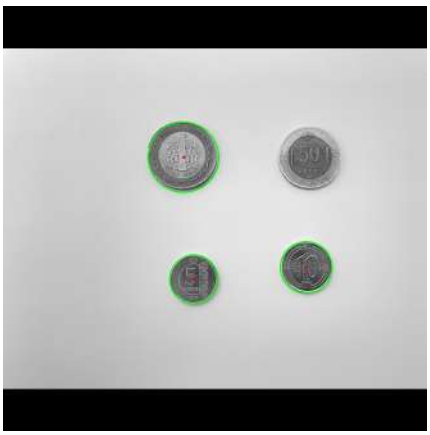
Undetected Parts

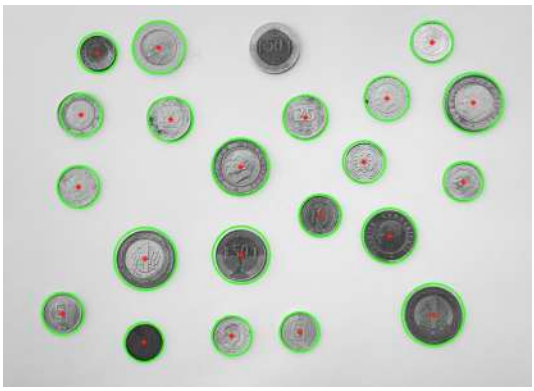
Inconsistent Edge Detection: If the edges of the coins are not distinctly different from the background, the Canny edge detector might not pick them up. The algorithm relies on finding sharp changes in intensity to identify edges. If the coins blend too much with the background or if the lighting is not sufficient, the edges may not be detected correctly.

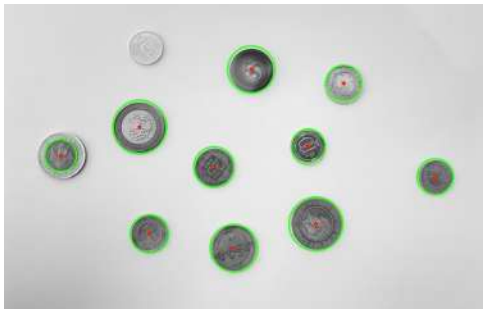
Adjustment of Parameters: The parameters chosen for edge detection and the Hough Transform could be too restrictive or too lenient. This could result in missing out on circles that do not meet the strict criteria or in detecting too many features, thus overlooking the correct ones.

Varying Coin Sizes: The radius range provided to the Hough Transform might not encompass the actual sizes of the coins. If the range is too narrow or does not include the radius of certain coins, those coins will not be detected.

Noise and Artifacts: Any noise or artifacts present in the image can create false edges or suppress the actual edges of the coins. This can also result in incorrect or missed circle detection.







PART 2

RoI part

Ensuring Valid Circle Parameters: The circle's center and radius are adjusted to ensure they fall within the image's boundaries.

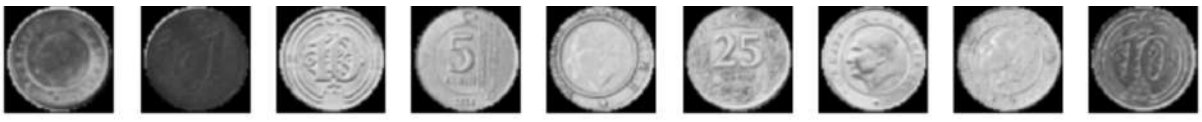
Mask Creation:A mask is created that has the same dimensions as the image. On this mask, a white circle is drawn where the detected circle is located, with the rest of the mask remaining black.

Applying the Mask: The mask is applied to the image using a bitwise 'AND' operation. This results in the area inside the circle being kept while the rest is discarded, effectively isolating the ROI.

Cropping and Resizing: The area within the mask is then cropped and resized to a standard dimension, ensuring uniformity across all ROIs for the HOG descriptor.

Handling of Empty ROIs:In cases where an ROI is empty, a message is displayed, indicating that the image or circle parameters may not have been set correctly.

Preparation for HOG: The resized ROIs are then ready to be used for HOG feature extraction. These ROIs provide the HOG descriptor with consistent areas of the image to analyze, enabling it to compute and describe the gradients effectively.



Example Of Rois

Histogram Part

First, the function cell_histogram is defined to create a histogram for each cell of the image. A cell is a small block of the image, and its size is determined by the cell_size parameter. The angles and magnitudes of the gradients in these cells are used to compute the histogram.

Here's how the cell_histogram function operates:

- A histogram is initialized with a number of bins based on the bin_size, which determines the range of angles each bin covers.
- The gradients of the image are then divided into cells.
- For each cell, the gradient magnitudes and angles are taken, and each angle is assigned to a bin in the histogram. The corresponding magnitude is added to that bin's count.
- This process is repeated for all the cells, resulting in a histogram that captures the frequency of different gradient orientations within the cell.

Next, the compute_histograms_for_gradients function takes the gradients for each Region of Interest (ROI) in an image and uses cell_histogram to compute a histogram for each one.

- For every ROI, the gradient magnitudes and angles are passed to the cell_histogram function.
- The computed histograms are collected for each image file name.
- These histograms represent the HOG features for each ROI and are stored in a dictionary.

The final step involves computing the HOG histograms for all ROIs in the training and test sets:

- The compute_histograms_for_gradients function is called with the gradient information for each set of ROIs.
- After the histograms are computed, a message is printed to indicate the completion of the process.

Normalization is first carried out using the normalize_histogram function, where each histogram's values are divided by its L2-norm. This step ensures that the histogram has a unit norm, which helps in reducing the effects of lighting and shadows in images. The normalize_histograms_list function applies this normalization to a list of histograms.

The normalize_histograms_per_image function extends this normalization to a collection of histograms associated with different images, ensuring that all histograms are on a consistent scale for comparison.

After normalization, the label_histograms function associates each histogram with a label derived from the image's filename, indicating the category of the image. This step is crucial for supervised learning, where each input must be paired with a correct output label.

Finally, to prepare for classification tasks, the histograms are flattened and converted into NumPy arrays, making them suitable for use with machine learning algorithms. If the dataset is imbalanced, with some labels having more samples than others, a downsampling process is conducted using the resample function. This process ensures that all categories have the same number of samples, leading to a balanced dataset that can improve the performance of classification algorithms.

```python
def label_histograms(normalized_histograms_per_image, filenames, circle_points):
    labeled_histograms = []
    labels = []
    count_undetected = 0

    for filename in filenames:
        # Skip the image if no circles were detected
        if len(circle_points[filename]) == 0:
            count_undetected += 1
            continue

        # Check if there are histograms for this file
        if not normalized_histograms_per_image[filename]:
            print(f"No histograms found for {filename}. Skipping...")
            continue

        # Extract the label from the filename
        lst = filename.split("_")
        cat = lst[0] + "_" + lst[1]

        # Assume each file has one histogram (adjust if there are multiple)
        histogram = normalized_histograms_per_image[filename][0]
        labeled_histograms.append(histogram)
        labels.append(cat)

    print("Number of undetected images: ", count_undetected, "out of ", len(filenames))
    return labeled_histograms, labels

# Label the normalized histograms for the train data
train_labeled_histograms, train_labels = label_histograms(train_normalized_histograms, train_filenames, train_circles)
```

Label Histogram

```python
from sklearn.utils import resample
import numpy as np

# Assuming train_histograms_array and train_labels_array are defined
# Example:
# train_histograms_array = np.array([...])
# train_labels_array = np.array([...])

# Find the minimum frequency among the labels
unique, counts = np.unique(train_labels_array, return_counts=True)
min_frequency = min(counts)

# Initialize lists to store the downsampled data
downsampled_histograms = []
downsampled_labels = []

# Downsample each class to the minimum frequency
for label in unique:
    # Get the indices of the current label
    indices = np.where(train_labels_array == label)[0]

    # Downsample the indices
    downsampled_indices = resample(indices, replace=False, n_samples=min_frequency, random_state=0)

    # Append the downsampled histograms and labels to the lists
    downsampled_histograms.extend(train_histograms_array[downsampled_indices])
    downsampled_labels.extend(train_labels_array[downsampled_indices])

# Convert the lists back to numpy arrays
downsampled_histograms_array = np.array(downsampled_histograms)
downsampled_labels_array = np.array(downsampled_labels)

# Now downsampled_histograms_array and downsampled_labels_array are balanced
```

Sampling the data

Firstly, the SVM classifier is created with a linear kernel and then trained using a set of downsampled histograms along with their corresponding labels. These histograms have been previously balanced across different classes to ensure that each class has the same representation in the training data.

```
]: # Flatten the histograms and extract them along with their labels into lists


# Now you can fit the SVM classifier
from sklearn.svm import SVC


svm_classifier = SVC(kernel='linear')


svm_classifier.fit(downsampled_histograms_array, downsampled_labels_array)
```

After the classifier is trained, it is used to classify objects within images. For each object, identified by the circles provided, the following steps are executed:

- A region of interest (ROI) is extracted from the image where the object is located.
- The gradients within this ROI are computed, and a histogram of these gradients is created.
- This histogram is then normalized to ensure consistency in the representation, regardless of variations in illumination or contrast.
- The classifier predicts the type of object based on this normalized histogram.

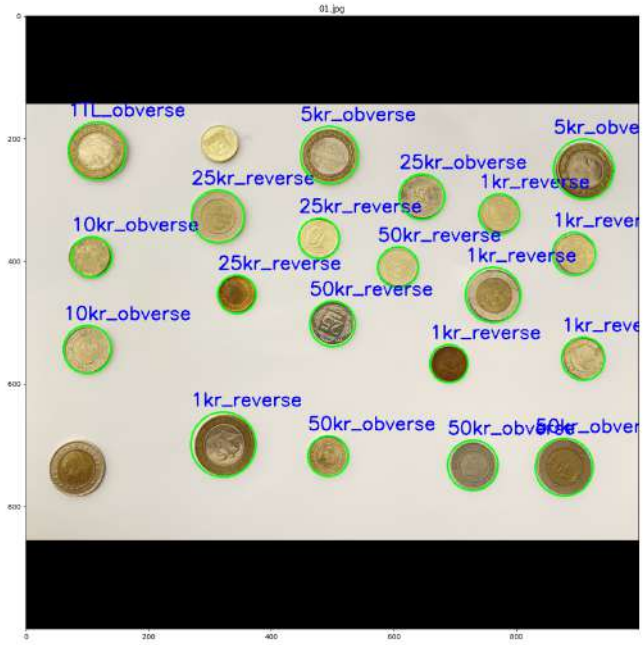
The predicted label and the location of the object are used to draw a circle and label the object directly on the image.

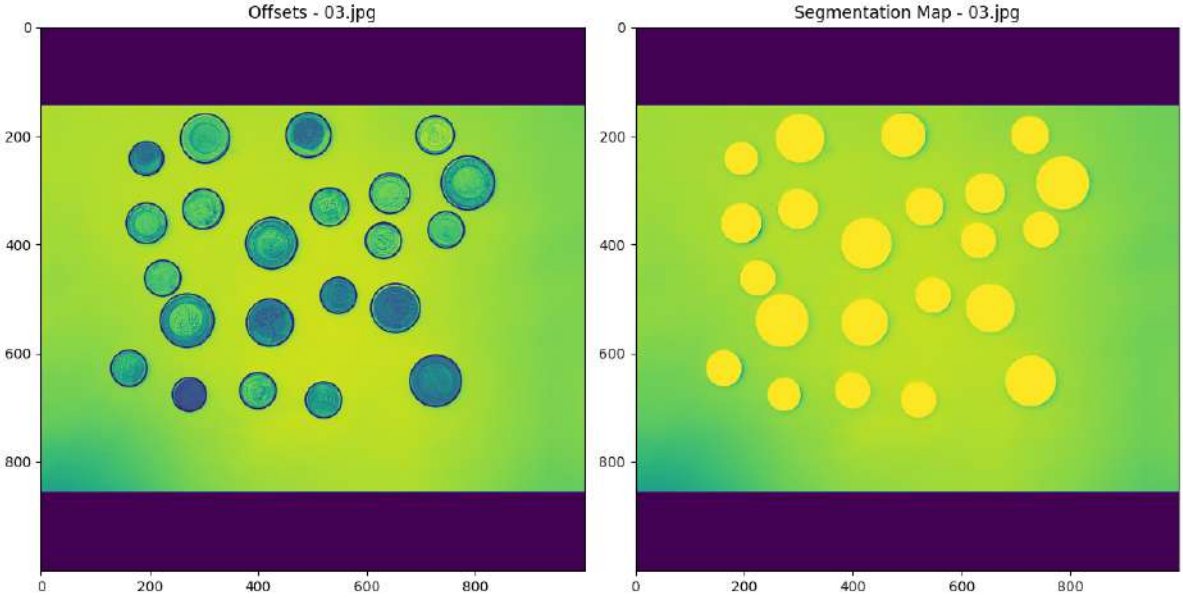
Subsequently, these images, now with the objects classified and marked, are saved to a specified directory. For visualization, the images are converted to the RGB color space, plotted with their corresponding labels, and then the plots are saved as image files.

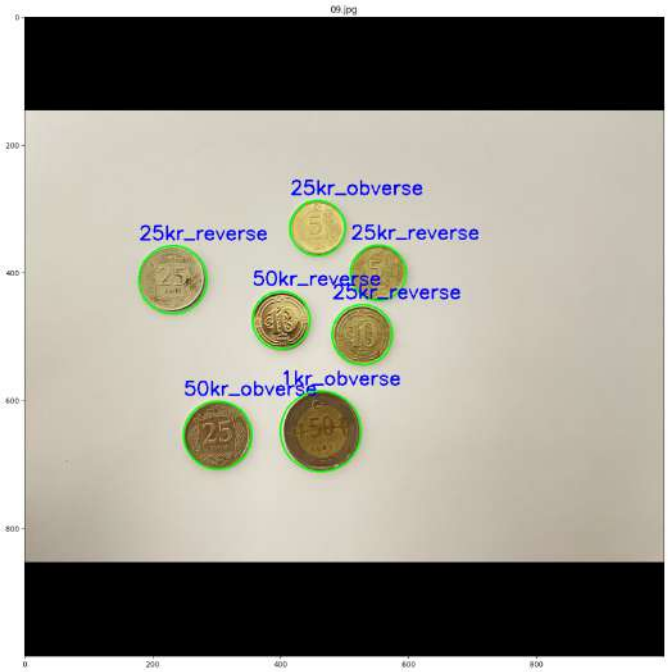
Additionally, a separate function is provided to plot offsets and create a segmentation map for each image, which is also saved to a designated output folder.

The entire process involves the following steps:

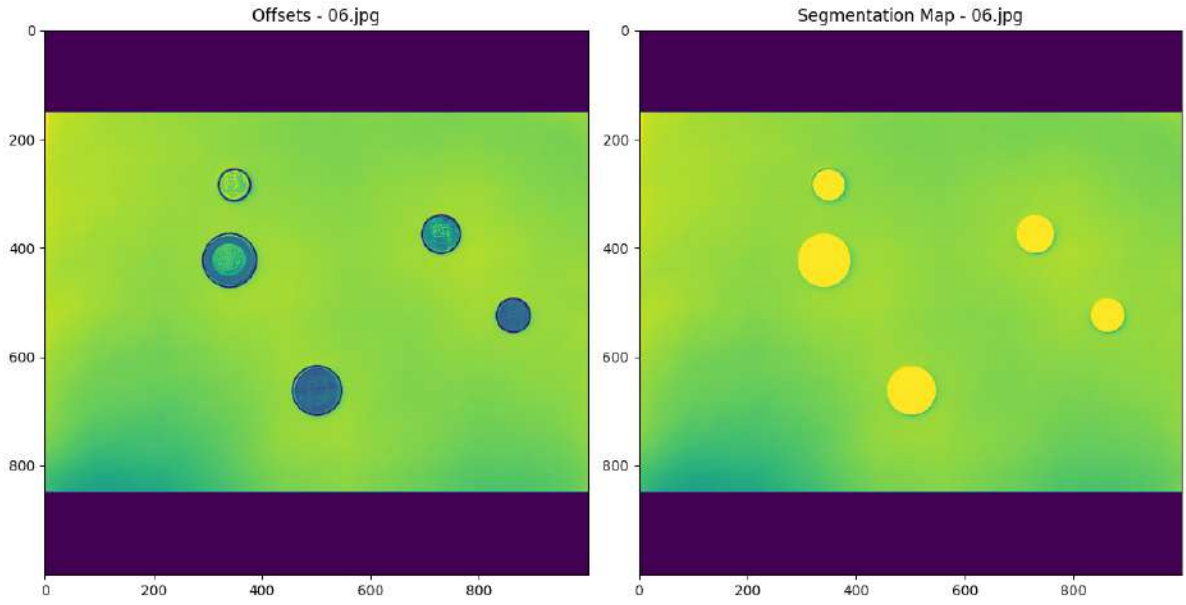
- Histograms are flattened and extracted.
- An SVM classifier is trained using the downsampled data.
- The trained classifier is then used to classify and label objects within new images.
- The results are visualized and saved for further analysis or inspection.


01.jpg
1L_obverse
5kr_obverse
5kr_obve
25kr_obverse
25kr_reverse
1kr_reverse
25kr_reverse
10kr_obverse
1kr_rever
50kr_reverse
1kr_reverse
25kr_reverse
50kr_reverse
10kr_obverse
1kr_reverse
1kr_reve
1kr_reverse
50kr_obverse
50kr_obver


Offsets - 03.jpg
Segmentation Map - 03.jpg
0
200
400
600
800



TestV_Hog Example



Failing

Feature Scale Variability: HoG features are sensitive to the scale at which they are computed. If the coins in the training set range widely in size, the gradients (which the HoG descriptors are based on) would capture different levels of detail for each coin. This can lead to a mismatch when the classifier encounters coins of sizes it hasn't effectively learned to recognize.

Inconsistent Region of Interest (ROI) Extraction: When coins of different sizes are present in the training data, the extracted ROIs may not be consistent. Smaller coins might lead to ROIs that include a lot of background information, while larger coins might have ROIs tightly cropped around the coin's edges. This inconsistency can confuse the classifier, which relies on uniformity in the feature set.

Resolution Discrepancy: If coins are not resized to a standard dimension before HoG feature extraction, the resolution of the coin images will vary. High-resolution images of large coins will contain more detailed information compared to low-resolution images of small coins, making it difficult for the classifier to extract and learn generalizable features.

Spatial Information Loss: Resizing images to a standard size as a solution to size variability can introduce another problem. It can distort spatial relationships within the HoG features if the aspect ratio is not maintained. This could lead to a loss of valuable shape and texture information, which is crucial for accurate classification.