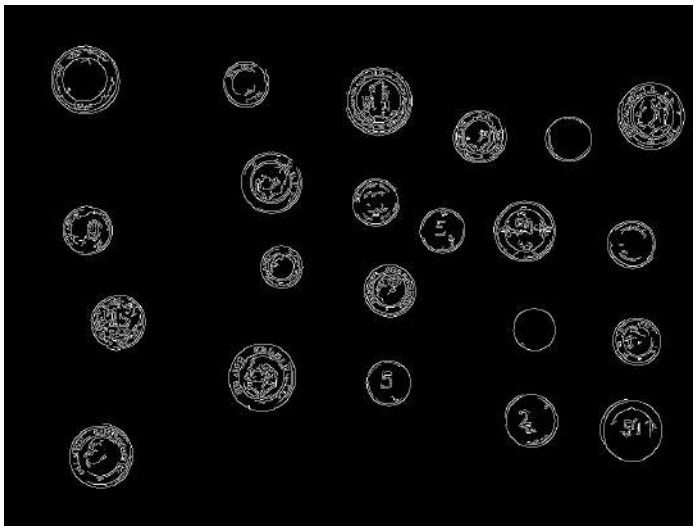
AIN 433 - Introduction to Computer Vision Laboratory
Programming Assignment 2
Fall 2023
Assoc. Prof. Dr. Ali Seydi KEÇELİ
R.A. Görkem AKYILDIZ
Student ;
Name: Sare Naz
Surname: Bayraktutan
ID: 21992957

Overview
The Hough transform is a feature extraction technique used in image analysis, computer vision, and digital image processing. The purpose of the technique is to find imperfect instances of objects within a certain class of shapes by a voting procedure. This voting procedure is carried out in parameter space, from which object candidates are obtained as local maxima in a so-called accumulator space that is explicitly constructed by the algorithm for computing the Hough transform. The aim of this project is to get used to Hough Transform and HoG features.

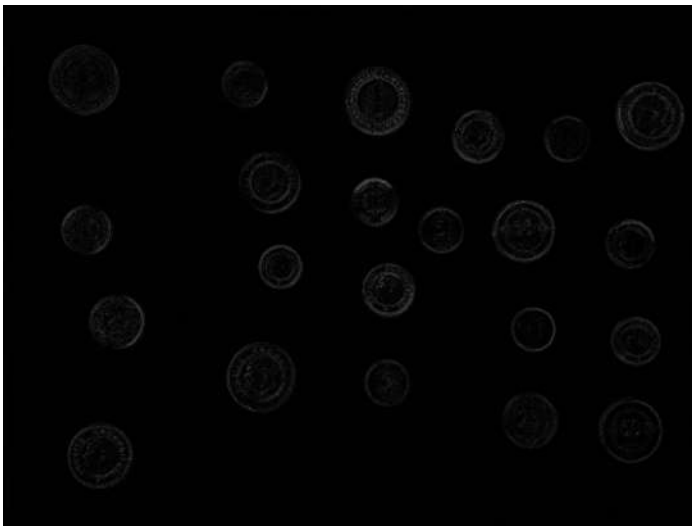
• Explain which edge detector you have chosen and your reasoning. Moreover, explain how you set the parameters of the edge detection algorithm. Show edge detection results in the document with the relevant code snippet you have written

For this assignment, Laplacian and Canny edge detectors were used, and it was observed that the best result was given by Laplacian. Initially, Canny was applied, and optimal parameters were researched on the internet and applied, resulting in a satisfactory outcome. Subsequently, Laplacian was applied. It provided a more effective result in Edge Detection, but due to its noticeably longer duration, Canny was utilized. No improvement was observed in the prediction of the detected edges due to the Edge Detector.

Canny Edge Detection Results:



Laplacian Edge Detection Results:

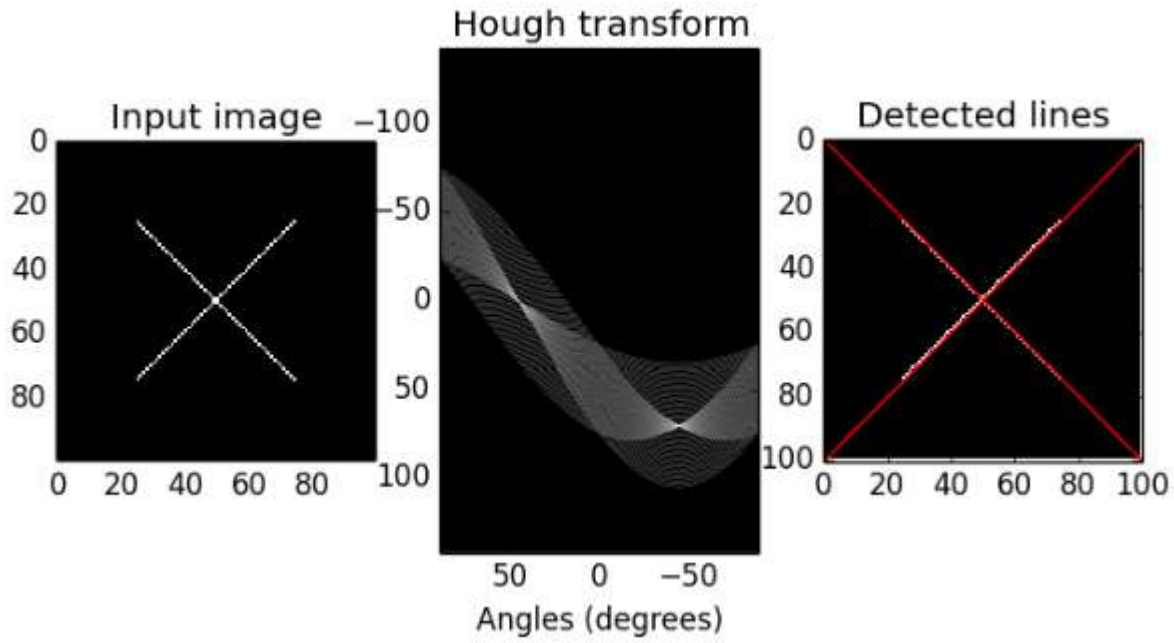


Code snippet for edge detection:

```python
def process_image(image_path, output_dir, edge_output_dir, distance_threshold):
    original_image = cv2.imread(image_path, cv2.IMREAD_COLOR)
    max_size, canny_thresholds, radius_range = (200, (150, 200), np.arange(30, 100)) if 'Train' in imag

    height, width = original_image.shape[:2]
    scale = min(max_size / height, max_size / width)
    if scale < 1:
        new_dimensions = (int(width * scale), int(height * scale))
        resized_image = cv2.resize(original_image, new_dimensions, interpolation=cv2.INTER_AREA)
    else:
        resized_image = original_image
    output_image = resized_image.copy()
    gray_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)
    blurred_image = cv2.GaussianBlur(gray_image, (3,3), 0)
    edges = cv2.Canny(blurred_image, *canny_thresholds)
```

```python
def process_image(image_path, output_dir, edge_output_dir, distance_threshold):
    original_image = cv2.imread(image_path, cv2.IMREAD_COLOR)
    max_size, radius_range = (200,  np.arange(30, 100)) if 'Train' in image_path else (1000,  np.arange

    height, width = original_image.shape[:2]
    scale = min(max_size / height, max_size / width)
    if scale < 1:
        new_dimensions = (int(width * scale), int(height * scale))
        resized_image = cv2.resize(original_image, new_dimensions, interpolation=cv2.INTER_AREA)
    else:
        resized_image = original_image
    output_image = resized_image.copy()
    gray_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)
    blurred_image = cv2.GaussianBlur(gray_image, (3,3), 0)

    # Apply Laplacian edge detection
    laplacian = cv2.Laplacian(blurred_image, cv2.CV_64F)
    edges = cv2.convertScaleAbs(laplacian)
```

• Explain all steps of Hough transform with visual examples and code snippets you have written.
The Hough Transform is a technique utilized for detecting lines in an image, proving particularly effective in scenarios where the lines are obscured or fragmented. The steps involved in its application are as follows:



- **Grayscale Conversion:** Conversion to grayscale is initially performed on the image, as the Hough Transform operates on intensity variations and does not necessitate color information.
- **Edge Detection:** An edge detection algorithm, such as the Canny edge detector, is applied to the grayscale image, serving to highlight potential lines by identifying edges.
- In the Hough Transform, the possible lines that might intersect each edge point are considered. While a line in image space can typically be represented as y=mx+c, this representation encounters issues with vertical lines. Therefore, the Hough Transform adopts a different representation: r=xcos(θ)+ysin(θ), where r denotes the distance from the origin to the closest point on the line, and θ represents the angle between the x-axis and the line connecting the origin to this nearest point.
- **Center Computation:** For a given point (x,y) and a specified radius and gradient direction, the potential center (a,b) of a circle passing through that point is computed.
- **Center Validation:** All calculated centers undergo validation checks against the bounds of the accumulator space, ensuring they reside within the acceptable range of the Hough space dimensions
- **Hough Transform Execution:** To detect circles, the Hough Circle Transform is applied to an image where edges have been detected. An accumulator space, configured as a 3D array, is set to zero. For every edge point and for each specified radius, potential circle centers are calculated and validated, with valid centers receiving votes in the Hough space.
- **Non-maximum Suppression:** Subsequent to the detection of potential circles, non-maximum suppression is utilized to eliminate nearly identical circles based on a set distance threshold. This process ensures the final list of detected circles is devoid of redundancies and accurately represents distinct physical circles in the image.
- **Circle Detection:** The centers and radii of the detected circles are ascertained by applying a threshold to the Hough space and extracting coordinates that have amassed a sufficient number of votes. These are then organized in accordance with the value of their accumulator, with higher values indicating a greater likelihood of representing actual circles.
- **Drawing Circles:** For visual representation, the detected circles are outlined on the original image. Two circles are drawn for each detected circle: one with a predetermined color to emphasize the edge, and another with a chosen color to delineate the boundary.
- I**mage Resizing and Output:** Should the original image surpass the maximum size limit, resizing is conducted. The detected circles are then superimposed on the resized image. Padding is computed and applied to preserve the aspect ratio, and the final image is stored in the designated output directory.
- This method can also be adapted to detect other shapes, such as circles or ellipses, by employing a parameter space suited to those particular shapes.

In [2]:
```python
#Computes the center (a,b) of a circle passing through point (x,y) with a given radius and angle of gradient.
def calculate_vote(x, y, radius, direc):
    a = int(x - radius * math.cos(direc))
    b = int(y - radius * math.sin(direc))
    return a, b
```

In [3]:
```python
#Checks if the calculated circle center (a,b) is within the bounds of the hough_space dimensions.
def validate_vote(a, b, hough_space_shape):
    return a >= 0 and a < hough_space_shape[1] and b >= 0 and b < hough_space_shape[0]
```

```python
# Processes an image by resizing it, detecting edges, finding circles using Hough Transform,
#applying non-maximum suppression, drawing the circles, padding the image to a square, and saving the results.

def process_image(image_path, output_dir, edge_output_dir, distance_threshold):
    original_image = cv2.imread(image_path, cv2.IMREAD_COLOR)
    max_size, canny_thresholds, radius_range = (200, (150, 200), np.arange(30, 100)) if 'Train' in image_path else (1000, (50, 1

    height, width = original_image.shape[:2]
    scale = min(max_size / height, max_size / width)
    if scale < 1:
        new_dimensions = (int(width * scale), int(height * scale))
        resized_image = cv2.resize(original_image, new_dimensions, interpolation=cv2.INTER_AREA)
    else:
        resized_image = original_image
    output_image = resized_image.copy()
    gray_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)
    blurred_image = cv2.GaussianBlur(gray_image, (3,3), 0)
    edges = cv2.Canny(blurred_image, *canny_thresholds)

    edge_image_name = os.path.splitext(os.path.basename(image_path))[0] + '_Edge.jpg'
    edge_image_path = os.path.join(edge_output_dir, edge_image_name)
    cv2.imwrite(edge_image_path, edges)

    sobel_x = cv2.Sobel(blurred_image, cv2.CV_64F, 1, 0, ksize=3)
    sobel_y = cv2.Sobel(blurred_image, cv2.CV_64F, 0, 1, ksize=3)
    mag, direc = np.sqrt(sobel_x**2 + sobel_y**2), np.arctan2(sobel_y, sobel_x)

    hough_space = hough_transform(edges, mag, direc, radius_range)
    threshold = 0.5 * hough_space.max()
    detected_circles = [(x, y, radius_range[r], hough_space[y, x, r]) for y, x, r in zip(*np.where(hough_space >= threshold))]
    nonmax_circles = nonmax_supp(detected_circles, distance_threshold)

    scale = min(max_size / height, max_size / width)

    if scale < 1:
        new_dimensions = (int(width * scale), int(height * scale))
        output_image = cv2.resize(original_image, new_dimensions, interpolation=cv2.INTER_AREA)
    else:
        output_image = original_image.copy()

    for x, y, radius, _ in nonmax_circles:
        cv2.circle(output_image, (x, y), radius, (0, 0, 256), 3)
        for x, y, radius, _ in nonmax_circles:
            cv2.circle(output_image, (x, y), radius, (0, 0, 256), 3)
    # Calculate and apply padding
    height, width = output_image.shape[:2]
    del_h, del_w = max_size - height, max_size - width
    pad_top, pad_bottom = del_h // 2, del_h - del_h // 2
    pad_left, pad_right = del_w // 2, del_w - del_w // 2
    pad_image = cv2.copyMakeBorder(output_image, pad_top, pad_bottom, pad_left, pad_right, cv2.BORDER_CONSTANT, value=(256, 256,

    cv2.imwrite(os.path.join(output_dir, os.path.basename(image_path)), pad_image)
```

- **Put samples of your detected circles as in Figure 2.**

The results of the Hough Transforms, applied following the Canny edge detection, were previously presented. The outcomes of applying the Hough Transforms after the Laplacian edge detection will now be demonstrated.

• Show parts that your method fails to detect circles correctly and explain possible reasons for it. If your code successfully works, you can omit this step by denoting that your code successfully works.

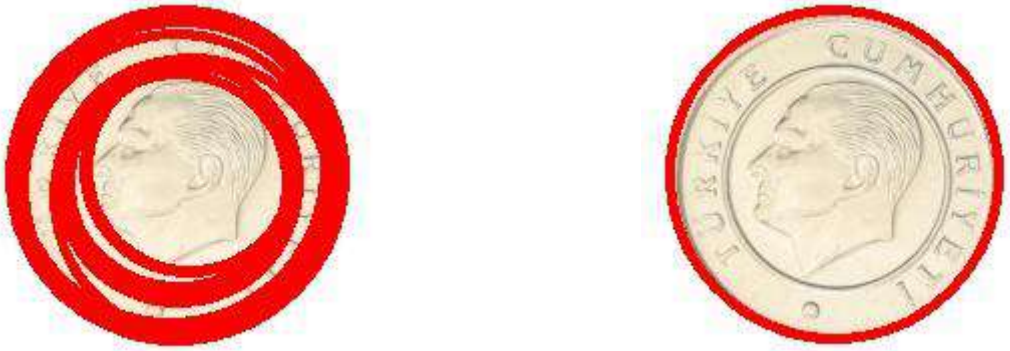
Several factors could be affecting the accuracy of circle detection using the Hough Transform, leading to incorrect results. Here are some potential reasons:

Parameter Selection: The process of detecting circles with the Hough Transform is highly sensitive to parameter settings. Parameters such as edge detection thresholds, the range of radii to be searched, and the threshold for the accumulator that decides which circles are significant enough to be reported, if not chosen correctly, can cause the omission of actual circles or the detection of false positives.
Edge Detection Quality: The quality of the initial edge detection is crucial for the success of the Hough Transform in detecting circles. If the Canny edge detector fails to accurately identify the edges of circles, perhaps because of noise, blur, or improper threshold values, then the Hough Transform based on these edges will also be inaccurate. A review of images processed using both Canny and Laplacian Edge Detection might show that, in the context of the Hough Transform, Laplacian Edge Detection could yield better results due to its different handling of edge features.

Accumulator Resolution: The precision of the circle detection is affected by the size of the bins in the accumulator array. Multiple circles might be merged into one detection if the resolution is too low; conversely, a single circle could be split into multiple detections if the resolution is too high.

Non-maximum Suppression Threshold: The extent to which two circles can be proximal before being considered duplicates is determined by the distance threshold used in non-maximum suppression. Distinct circles may be erroneously merged if this threshold is set too large; alternatively, if the threshold is set too small, the same circle might be detected multiple times.

Without Non-Max Suppression: With Non-Max Suppression:



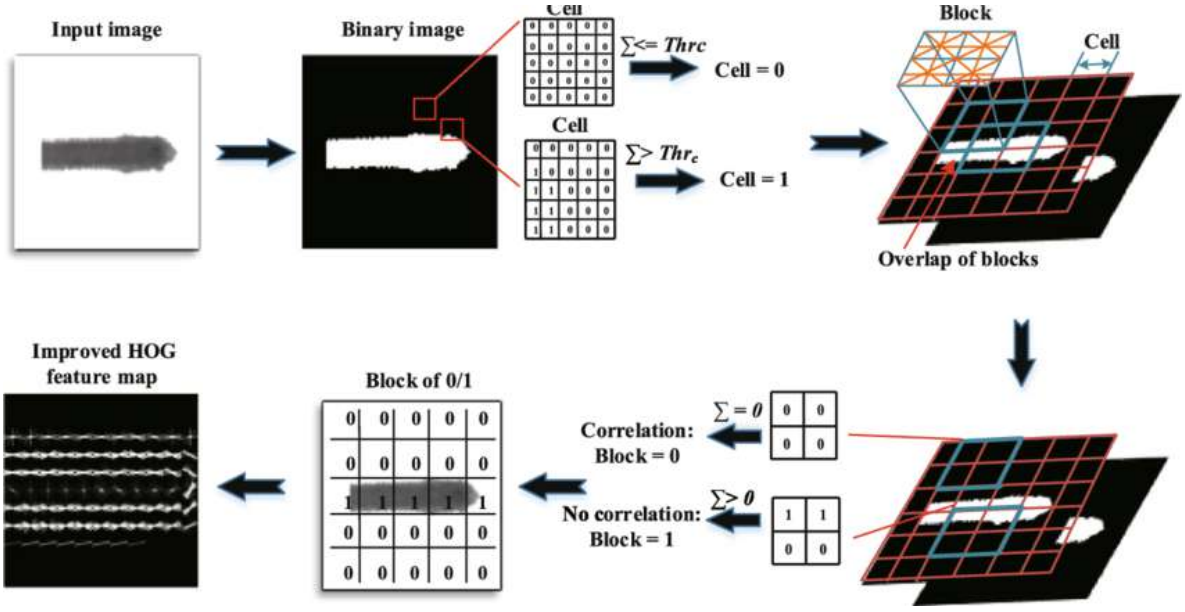
Noise and Artifacts: Spurious edges may be caused by excessive noise and artifacts in the image, which the Hough Transform could misinterpret as circles.

Limited Variety: The full range of variations in circle appearance, size, color, and occlusion levels that occur in real-world situations may not be represented by a small dataset. Consequently, circles that do not fall within the narrow range represented in the training set may not be detected by the method.

Overfitting: Overfitting to a small dataset can occur, with the model learning to detect circles solely under the specific conditions present in the training data. This may lead to poor generalization when confronted with new images that vary in appearance

• Explain all steps of the HoG Algorithm with visual examples and code snippets you have written.

The Histogram of Oriented Gradients (HoG) algorithm is characterized by several steps, which are typically executed in the following sequence:



Preprocessing: Initially, the image is subjected to preprocessing where contrast and brightness normalization are conducted to enhance the robustness of subsequent edge orientation detection.
Gradient Computation: Subsequently, gradients are computed across the image, with the horizontal (Gx) and vertical (Gy) components often determined using Sobel operators.
Orientation Binning: The orientation of gradients is ascertained through the calculation of the arctangent of the ratio of Gy to Gx for each pixel. These orientations are allocated to bins, with the range of angles usually segmented into uniform parts representing the bins.
Descriptor Blocks Assembly: Cells are grouped into larger, spatially connected blocks. Within these blocks, the histograms are normalized, which provides better invariance to changes in illumination and shadowing.
Normalization: The histograms are normalized across the block, significantly reducing the influence of lighting variations.
Feature Vector Construction: The normalized histograms are concatenated to form the final feature descriptor.

```
]: #Creates histograms of gradient orientations within cells, normalizes them in bl
   #the HOG feature vector.
   def create_and_normalize_histograms(mag, direc, cell_size=32, bin_size=9, block_
       cell_rows, cell_cols = mag.shape[0] // cell_size, mag.shape[1] // cell_size
       histograms = np.zeros((cell_rows, cell_cols, 180 // bin_size))
       val = 1e-7  # Small value to avoid division by zero
       normalized_histograms = []
       for i in range(cell_rows):
           for j in range(cell_cols):
               cell_slice = (slice(i * cell_size, (i + 1) * cell_size), slice(j * c
               cell_mag = mag[cell_slice]
               cell_direc = direc[cell_slice]
               histograms[i, j] = np.histogram(cell_direc, bins=np.arange(0, 181, b
       for i in range(histograms.shape[0] - block_size + 1):
           for j in range(histograms.shape[1] - block_size + 1):
               block = histograms[i:i+block_size, j:j+block_size].flatten()
               normalized = block / np.sqrt(np.sum(block**2) + val)
               normalized_histograms.append(normalized)

       return np.concatenate(normalized_histograms)
```

```
]: #calculates gradient magnitudes and directions, then extracts Histogram of Orien
   #features using these gradients.
   def calculate_hog(image, cell_size=32, bin_size=9, block_size=3):
       grad_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
       grad_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

       mag = np.hypot(grad_x, grad_y)
       direc = np.degrees(np.arctan2(grad_y, grad_x)) % 180

       hog_features = create_and_normalize_histograms(mag, direc, cell_size=cell_si
       return hog_features
```

```
hog_features = calculate_hog(processed_region_of_interest).reshape(1, -1)
```

• **Put samples of your detected coins with a description.**

TestV:

TestR:


50kr_reverse
50kr_reverse
25kr_reverse
50kr_obverse
1kr_obverse
50kr_reverse
50kr_obverse
25kr_obverse
50kr_obverse
1TL_obverse
1kr_obverse
25kr_obverse
10kr_obverse
50kr_obverse
25kr_obverse
25kr_reverse
1kr_obverse
50kr_reverse
1TL_obverse


25kr_reverse
50kr_reverse
25kr_obverse
50kr_obverse
25kr_reverse
50kr_reverse
50kr_obverse
1kr_obverse
25kr_obverse
50kr_reverse
50kr_obverse
50kr_obverse
1TL_obverse
25kr_obverse
50kr_obverse

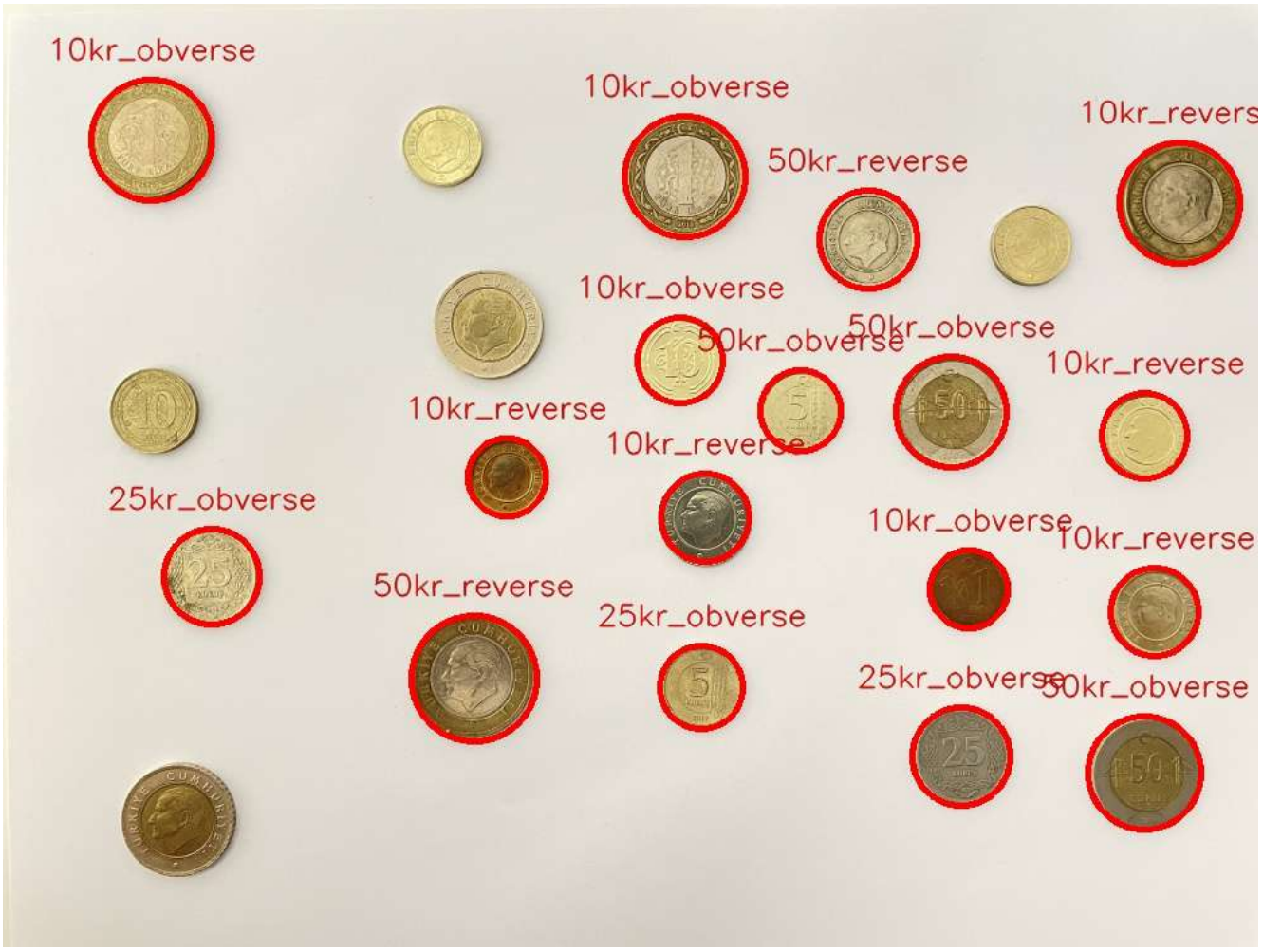
• Show parts that your HoG Classification fails to recognize coins and explain possible reasons for it.

Small Dataset: A small dataset may not provide sufficient variability in coin appearances. This lack of diversity can prevent the classifier from learning the necessary features to accurately distinguish between different types of coins.
Inconsistent Coin Proportions: If the dataset contains coins with varying proportions or sizes and does not include adequate representation of these variations, the classifier might fail to recognize coins when their size or appearance in the image differs significantly from the examples in the training set.
The HoG classifier might struggle with ratio-based detection, particularly if the dataset includes coins of varying sizes and proportions. This limitation becomes more pronounced if the classifier has been trained on a dataset where coins are consistently presented in a certain size or orientation, leading to difficulties in generalizing to coins of different sizes or orientations in new images.

Orientation Sensitivity: The classifier's ability to recognize coins may be compromised when coins are presented upside down, as observed in the TestR dataset. This indicates a sensitivity to the orientation of the coins, where the classifier struggles to identify them if they are not oriented as they were in the training examples. This challenge arises when the training data does not include a diverse range of orientations for each coin type, limiting the classifier's ability to generalize to different orientations in new images.

EXTRA PART

In this part, the results of the k-Nearest Neighbors (kNN) k=3 algorithm will be presented. This was done for the purpose of experimentation and to observe any differences in the results.


10kr_obverse
10kr_obverse
50kr_reverse
10kr_obverse
50kr_obverse
50kr_obverse
10kr_reverse
10kr_reverse
10kr_reverse
25kr_obverse
10kr_obverse
10kr_reverse
50kr_reverse
25kr_obverse
25kr_obverse
50kr_obverse


50kr_obverse
10kr_reverse
10kr_reverse
10kr_obverse
10kr_obverse
10kr_reverse
1TL_obverse
10kr_obverse
10kr_obverse
10kr_reverse
10kr_obverse
50kr_reverse
10kr_reverse
50kr_obverse
10kr_obverse

In conclusion, it was observed that the k-Nearest Neighbors (kNN) algorithm identified nearly half of the coins as 10 kr and the other half as 50 kr. This is thought to be due to a lack of sufficient data. Similar results were provided when k was set to 5. An increase in the value of k was not tested further, as there were only five examples for each side of every coin in the dataset.

RESOURCES

https://www.analyticsvidhya.com/blog/2022/06/a-complete-guide-on-hough-transform/#:~:text=Hough%20Transform%20is%20a%20computer,they're%20broken%20or%20obscured.
https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html
https://docs.opencv.org/4.x/da/d53/tutorial_py_houghcircles.html
https://medium.com/@isinsuarici/hough-circle-transform-in-opencv-d74bdf5161ed
https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html
https://github.com/PavanGJ/Circle-Hough-Transform/blob/master/main.py
https://github.com/adityaintwala/Hough-Circle-Detection
https://towardsdatascience.com/non-maxima-suppression-139f7e00f0b5
https://sharky93.github.io/docs/gallery/auto_examples/plot_line_hough_transform.html
https://thepythoncode.com/article/hog-feature-extraction-in-python
https://github.com/ahmedfgad/HOGNumPy
https://www.researchgate.net/figure/The-process-of-the-improved-HOG-algorithm-First-we-applied-the-HOG-algorithm-to-convert_fig1_326466337