AIN433 – Computer Vision Lab.

Programming Assignment 2

Burak Kurt

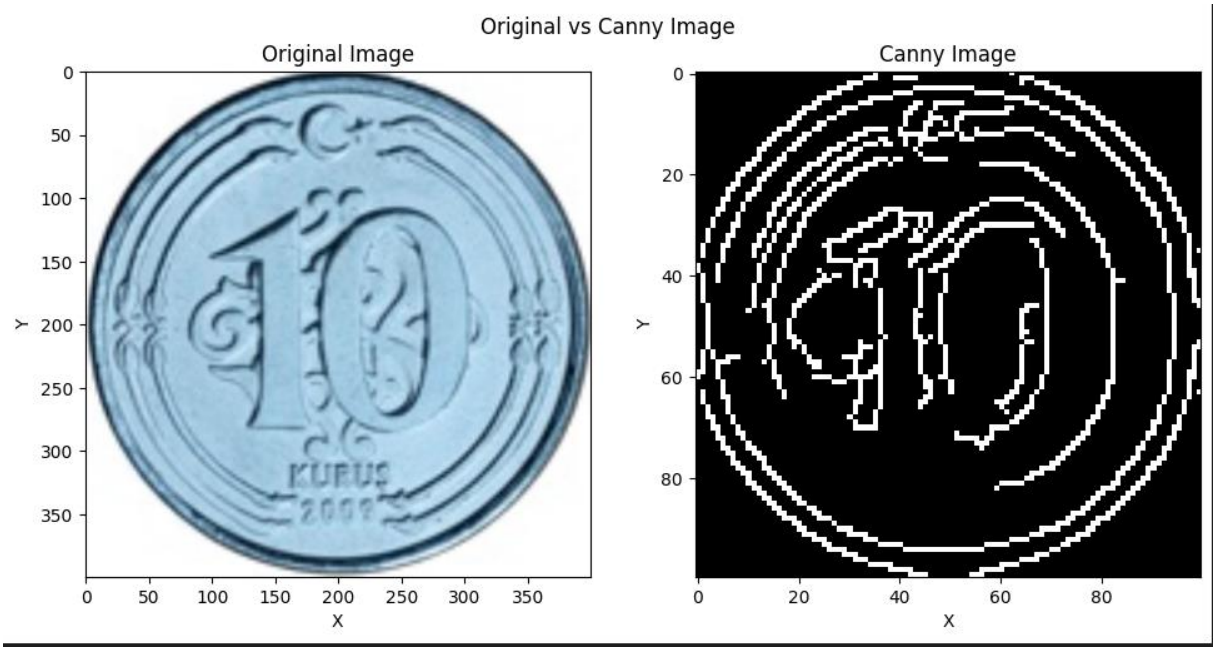
2200765010

PART 1: Edge Detection and Hough Transform:

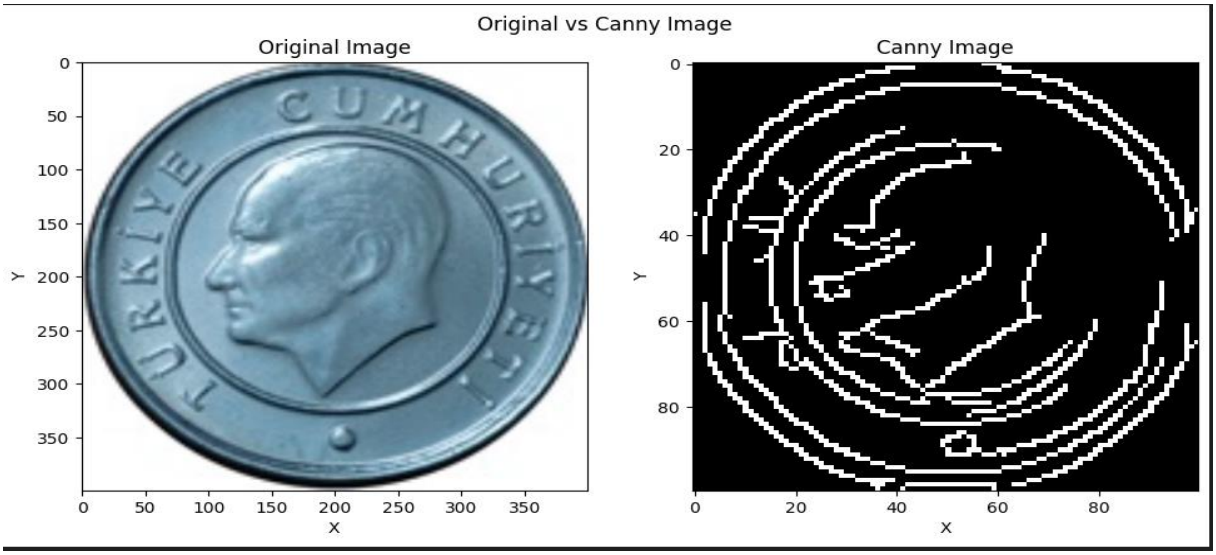
The main task in the first part of the assignment is to implement the Hough Circle Transform algorithm and detect possible circles on the given image. To be able to process images in this algorithm, preprocessing is needed which includes smoothing and edge detection. Smoothing is essential most of the time because images have noises in their original form and smoothing techniques reduce this noise and help to find “important” edges during edge detection. Built-in GaussianBlur function is used from OpenCV to apply smoothing on images with parameters of kernel size = (5,5) and sigma = 3. These parameters suit for training images with size 100x100 and test images with size 300x200. The canny edge detector algorithm is used for edge detection because parameter setting is easier than the Sobel algorithm. The optimal thresholds for 100x100 images are (50,150), and for 300x200 images are (0,15). These values are obtained from observations with different threshold values. After edge detection, images are ready to fit in the Hough Circle Transform algorithm.

```python
def applyCanny(image_array,l_Threshold,h_threshold):

    canny_Array = np.zeros((image_array.shape[0],image_array.shape[1],image_array.shape[2])) # 3D array
    for i in range(len(image_array)):
        gauss_img = cv2.GaussianBlur(image_array[i],(5,5),3)  # Gaussian Blur before edge detection
        canny_Array[i] = cv2.Canny(gauss_img,l_Threshold,h_threshold) # Canny Edge Detection
    return canny_Array
```

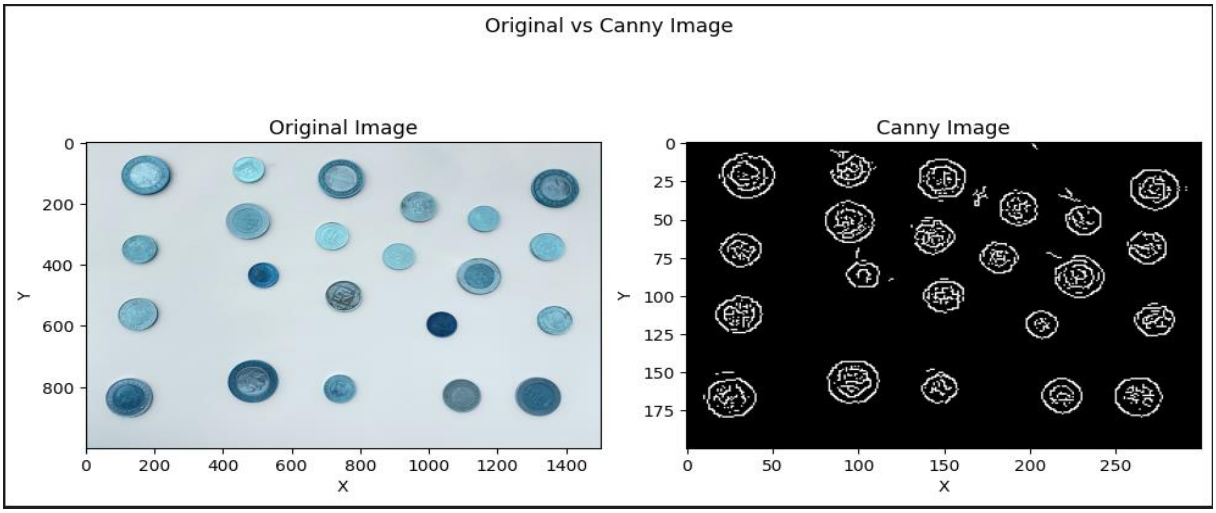
Canny Edge Detection Algorithm



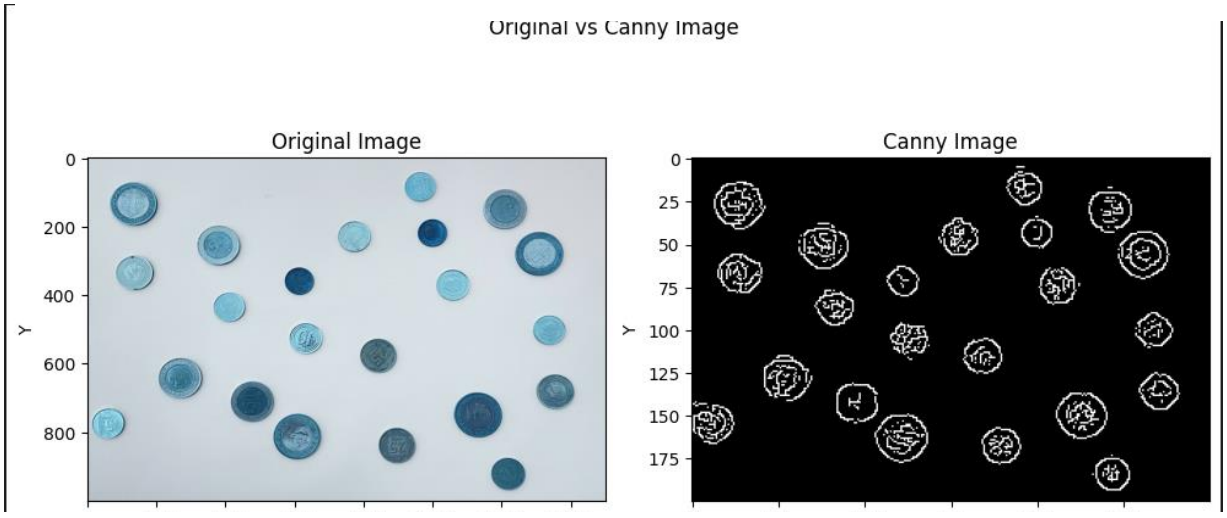
Example 1



Example 2



Example 3



Example 4

Hough Circle Transform algorithm takes an image in x-y space as input and converts it into a-b-r space where a and b are the center coordinates of the circle and r is the radius. In an implementation, a canny version of the image is given to the function. Then non-zero x and y coordinates are calculated where non-zero coordinates indicate edges. Since gradient orientation is unknown, sine and cosine values for all angles between 0 and 360 degrees are calculated. Then for all possible r values which are given as function parameter, the algorithm calculates the circle center for each edge point and store it in an array. After calculations, circle centers that have higher votes than the threshold are returned with their radius values from a function. The implementation also includes a circle number threshold which decides how many circles will remain after circle detection.

```python
def hough_circle_transform(image,min_rad,max_rad,threshold,circle_num):

    y,x = np.nonzero(image) # Pixel coordinates of edges

    # Accumulator array to store votes
    accumulator_arr = np.zeros((max_rad,image.shape[0],image.shape[1]))

    theta = np.arange(0,2*np.pi,np.pi/180)

    sin = np.sin(theta)
    cos = np.cos(theta)


    for i in range(len(x)):

        for r in range(min_rad,max_rad): # Since radius is unknown, loop is created for each radius value.

            a = (x[i] - r*cos)
            b = (y[i] - r*sin)
            a = a.astype(int)
            b = b.astype(int)


            valid = np.where((a>=0) & (a < image.shape[1]) & (b>=0) & (b < image.shape[0]))

            a = a[valid]
            b = b[valid]

            # Vote
            accumulator_arr[r,b,a] += 1


        circles = np.where(accumulator_arr > threshold) # Find circles with votes greater than threshold
        while(len(circles[0]) > circle_num): # If number of circles is greater than circle_num, increase threshold

            circles = np.where(accumulator_arr > threshold)
            threshold+=1


    radius = circles[0]
    x = circles[1]
    y = circles[2]

    return list(zip(radius,x,y))
```

Implementation of Hough Circle Transform algorithm

Example 1

Example 2

Example 3

The algorithm with given thresholds is successful in most of the training images but can not identify all the circles in the test images. The main reason behind this is the shape of the test images. Sizes of 300x200 are used for test images however not all the test images have a 3:2 ratio. When images with different ratios are resized into 300x200, some of the circles are transformed into ellipses and the algorithm could not find these shapes.

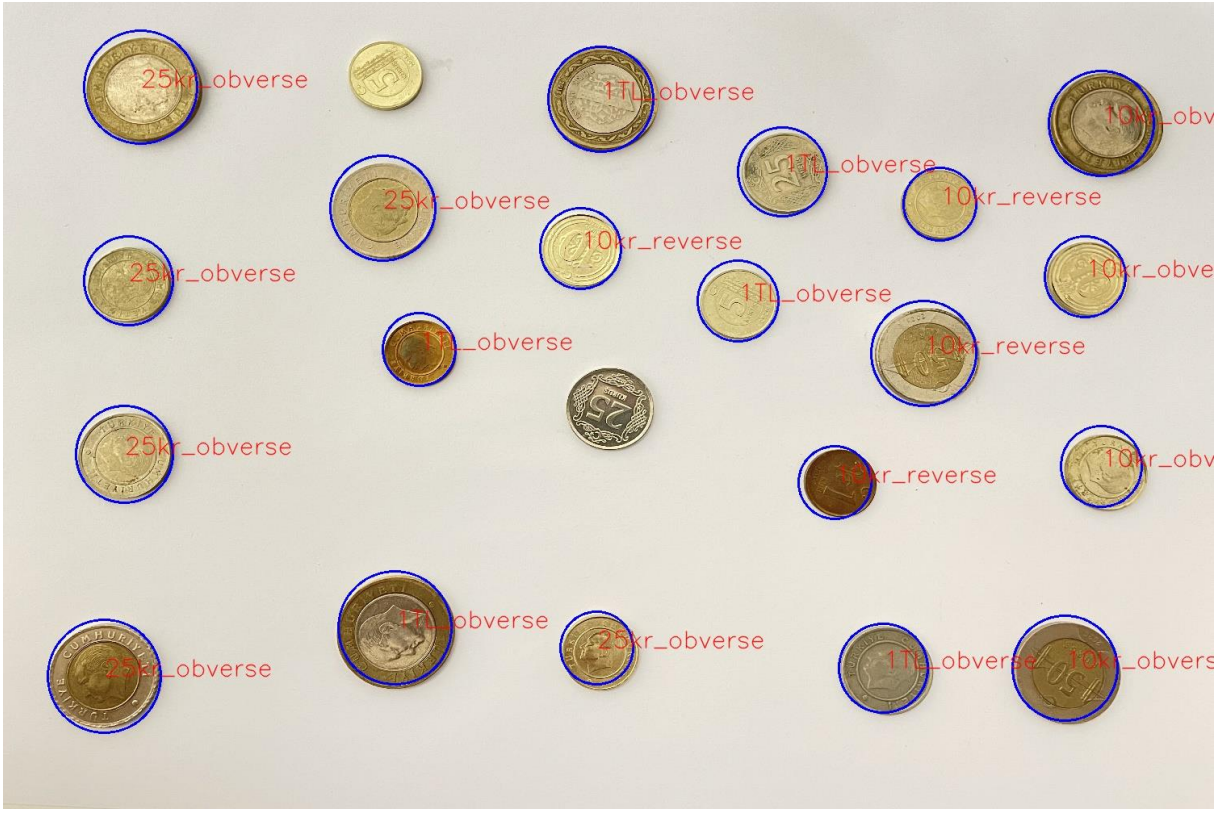
PART 2: Histogram of Oriented Gradients:

The HOG algorithm is a feature descriptor that can be used for object detection in images. It works by dividing the image into small regions called cells and computing a histogram of the gradient orientations for the pixels within each cell. The gradient orientation is the direction of the largest change in intensity in the image, and it can indicate the presence of edges, corners, or textures. The histograms are then normalized by using a larger region called a block, which helps to reduce the effects of illumination and shadowing. The normalized histograms are concatenated into a feature vector that represents the local shape and appearance of the object.

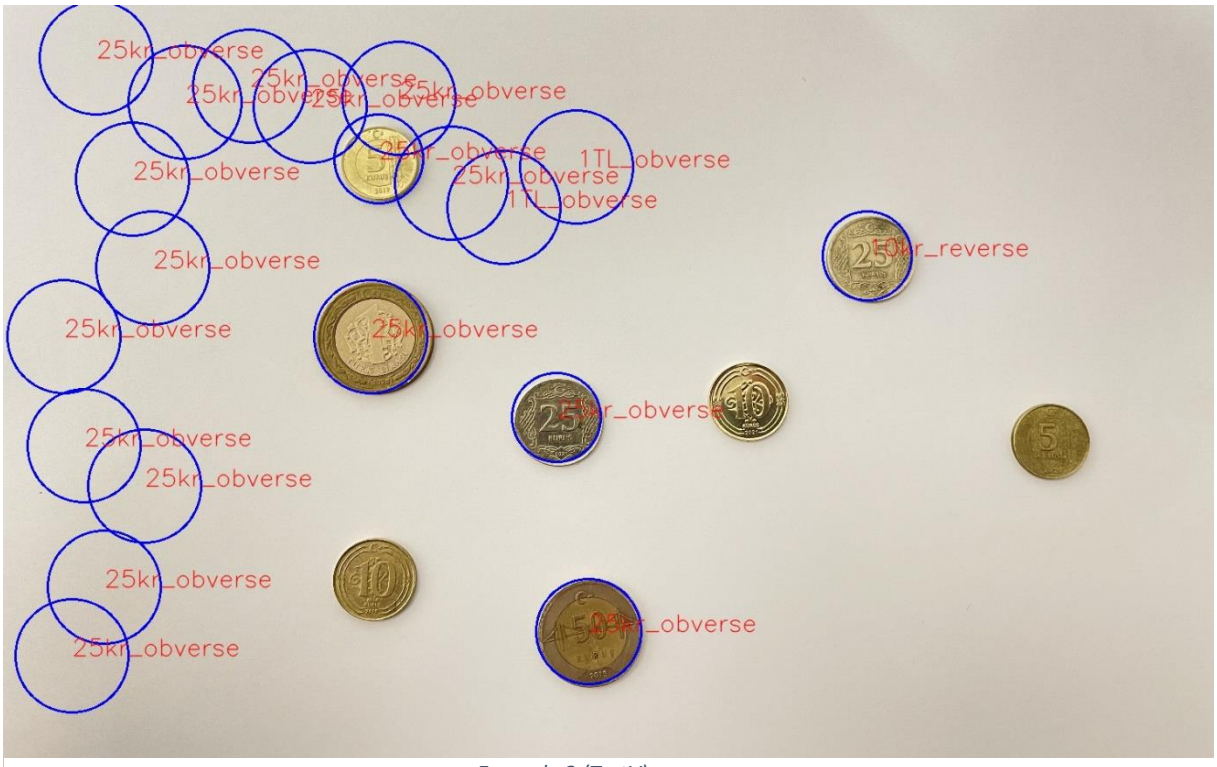
After training the SVM model with HOG features, circles in test images are cropped and given into the model. It is hard to measure how accurate the model is because the ground truth of test images is not known but the model's overall performance can be seen from images. Most of the predictions are assigned to "25kr_obverse" whether the coin is 25 kr or not. The main reason behind this result could be scaling. Test images are in size of 300x200 and detected circles are in size of 20x20 at most. 128x128 images are given to the HOG feature calculation function, so these circles have to be expanded before being given to HOG, so it causes information lost.

```python
def calculateHOG(image_gray):
    cell_size = 8
    block_size = 2
    bin_num = 9

    x,y = image_gray.shape

    image = image_gray.flatten()

    n_cells_x = int(x // cell_size) # number of cells in x direction
    n_cells_y = int(y // cell_size) # number of cells in y direction

    n_blocks_y = n_cells_y - block_size + 1
    n_blocks_x = n_cells_x - block_size + 1

    x_filter = np.array([-1,0,1])
    y_filter = x_filter.T

    gx = (np.convolve(image,x_filter,mode="same")).reshape(x,y) # gradient in x direction
    gy = (np.convolve(image,y_filter,mode="same")).reshape(x,y) # gradient in y direction



    magnitude = np.sqrt(gx**2 + gy**2)
    orientation = np.arctan2(gy,gx) * (180/np.pi) # orientation in degrees

    orientation  = orientation % 180 # orientation in range 0 to 180

    bin_edges = np.linspace(0,180,bin_num + 1) # bin edges for histogram
    #bin_centers = (bin_edges[1:] + bin_edges[:-1]) / 2

    hog_cells = np.zeros((n_cells_y,n_cells_x,bin_num))

    for i in range(n_cells_y):
        for j in range(n_cells_x):
            cell_mag = magnitude[i*cell_size : (i+1) * cell_size , j*cell_size : (j+1) * cell_size] # magnitude of cell
            cell_ori = orientation[i*cell_size : (i+1) * cell_size , j*cell_size : (j+1) * cell_size] # orientation of cell

            cell_bins = np.digitize(cell_ori,bin_edges)  # bin number of each pixel in cell

            for k in range(1,bin_num +1 ):
                hog_cells[i,j,k-1] = np.sum(cell_mag[cell_bins == k]) # sum of magnitude of pixels in each bin


    hog_blocks = np.zeros((n_blocks_y,n_blocks_x,block_size ** 2 *bin_num)) # hog features of each block
    for i in range(n_blocks_y):
        for j in range(n_blocks_x):
            block_hog = hog_cells[i:i+block_size,j:j+block_size,:]
            block_hog = block_hog.ravel() # flatten the block hog features

            block_hog = block_hog / np.sqrt(np.sum(block_hog**2) + 1e-6) # normalizing the block hog features

            hog_blocks[i,j,:] = block_hog


    hog_features = hog_blocks.ravel()

    return hog_features
```

HoG Algorithm



Example 1 (Testo)



Example 2 (TestV)