The task given to me is to eliminate duplicate images in the given data set. It is important to look for duplicates in the dataset as they affect the performance of the model to large extent. Duplicate images make the model remember features instead of finding the patterns in the dataset. Duplicate images provide more ways for the network to remember features instead of learning and truncating the model's ability to generalize. It also reduces the model ability to classify or detect an object during the test time.

Detecting similar images using a manual approach is hard and more time-consuming. It is also prone to human errors. In many cases, humans introduce bias as well. So, manual inspection of the dataset is to be avoided.

To do this, we need to perform operations based on the image contours area. The code is formatted in the way explained below.

i. Importing Libraries
ii. Defining functions to perform preprocessing steps on Images
iii. Helper Functions
iv. Function to calculate minimum contour areas
v. Loop to compare 2 images iteratively to remove if any duplicates are there in the directory.

Importing Libraries

```
import numpy as np
import cv2
import os
import glob
import matplotlib.pyplot as plt
import imutils
from tkinter import *
from tkinter import filedialog
import tkinter
from random import randint
from tqdm import tqdm
from time import sleep
```

The first step to perform any task is to import all required libraries into the program. Here we mainly rely on OpenCV, imutils, os, tkinter and NumPy libraries.

Defining functions to perform preprocessing steps on Images

```python
def draw_color_mask(img, borders, color=(0, 0, 0)):
    h = img.shape[0]
    w = img.shape[1]

    x_min = int(borders[0] * w / 100)
    x_max = w - int(borders[2] * w / 100)
    y_min = int(borders[1] * h / 100)
    y_max = h - int(borders[3] * h / 100)

    img = cv2.rectangle(img, (0, 0), (x_min, h), color, -1)
    img = cv2.rectangle(img, (0, 0), (w, y_min), color, -1)
    img = cv2.rectangle(img, (x_max, 0), (w, h), color, -1)
    img = cv2.rectangle(img, (0, y_max), (w, h), color, -1)

    return img
```

```python
def preprocess_image_change_detection(img, gaussian_blur_radius_list=None, black_mask=(5, 10, 5, 0)):
    gray = img.copy()
    gray = cv2.cvtColor(gray, cv2.COLOR_BGR2GRAY)
    if gaussian_blur_radius_list is not None:
        for radius in gaussian_blur_radius_list:
            gray = cv2.GaussianBlur(gray, (radius, radius), 0)

    gray = draw_color_mask(gray, black_mask)

    return gray
```

There are three pre-defined functions that are given to me in hand that perform operations on Images. First, an RGB image is converted into a Grayscale image. We apply Gaussian Blur to remove the noise in the image. The value of radius is selected after performing some experiments on different values and on multiple images. Then in each image, because of camera position, there is unwanted information on the left, right and top borders of the image. To eliminate this unnecessary information, instead of cropping the image and disturbing image spatial dimensions a simple technique of zero-pad is applied on the borders of all images.

Helper Functions

I have defined few helper functions for the program. First, a function to load all images in the data directory to a list.

```
def images_from_directory(data_dir):
    global images
    images = []
    files = [file for file in os.listdir(data_dir) if file.endswith('.png')]

    # reads all image files in .png format and convert them to RGB

    for i in range(len(files)):
        img = cv2.imread(str(files[i]),1)
        img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
        images.append(img)

    return images,files
```

This function takes in all files that are in .png format. Outputs of the function are images and filenames. Filenames helps to remove duplicate images as we will have names.

Next a function to convert all Images into GrayScale and apply Gaussian Blur to all the grayscale images. Output of the function is a list of images preprocessed.

```
def image_to_gray(images,gaussian_blur_radius_list):

    gray_images = []

    for i in range(len(images)):

        img = images[i].copy()

        gray = preprocess_image_change_detection(img,gaussian_blur_radius_list = gaussian_blur_radius_list)

        gray_images.append(gray)

    return gray_images
```

Now we have images preprocessed. The next step will be to compute contour areas of all contours that occur in an image.

```
def image_contour(image):

    thresh = cv2.adaptiveThreshold(image.copy(),255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY_INV,21,21)

    #plt.imshow(thresh)

    cnts = cv2.findContours(thresh,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)

    cnts = imutils.grab_contours(cnts)

    areas = []

    for c in cnts:
        area = cv2.contourArea(c)
        # threshold of 10 is set for the area to exclude small areas
        if area > 10:
            areas.append(area)

    average = np.mean(areas)

    return average
```

Each Image has different brightness ranges. It is hard to select the value that can satisfy all images. So, we use Adaptive Threshold here. The size of the patch for the computing threshold is 21x21. Following thresholding, we find contours in the image and grab all the contours in the image. Here I have calculated the area of each contour, but I have considered only the contours that have an area of more than 10. This is because, in the image, there are many small contours that are detected. In the image, you can see the statistics regarding the small areas occurring in the image.

```
small_areas = []
for i in range(len(areas)):
    if 0<areas[i]<=10:
        small_areas.append(areas)
```

From the stats it is clear that more than 40% of areas are small than 10 so excluding them helps for fast calculation.

```
print(len(small_areas))
small_areas_percent = len(small_areas)/len(areas)
print(small_areas_percent)
```

```
1440
0.4445816610064835
```

More than 40% of areas are in range 0 to 10. By excluding those areas, we can improve the computational speed. So, the value is taken from few observations.

Function to calculate minimum contour areas

As we are considering minimum contour areas, it is important to calculate them before we start comparing two images. This is because consider an example where we have 5 images in the directory. If we are comparing these 5 images, if we don't calculate the contour areas before then we must calculate the same contours again and again which is computational burden, and this slows down the program.

```
def image_contour(image):

    thresh = cv2.adaptiveThreshold(image.copy(),255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.THRESH_BINARY_INV,21,21)

    #plt.imshow(thresh)

    cnts = cv2.findContours(thresh,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)

    cnts = imutils.grab_contours(cnts)

    areas = []

    for c in cnts:
        area = cv2.contourArea(c)
        # threshold of 10 is set for the area to exclude small areas
        if area > 10:
            areas.append(area)

    average = np.mean(areas)

    return average
```

The final step is to take the predefined function and compare two images. To compare a single image to all other images it is also important to check if we are comparing same images more than once. So, it has been checked and no two images are compared more than once. Also, same image is not compared.

```
#initializing the lists
comp_list = []
scores = []

for i in tqdm(range(len(gray_images)-1)):

    for j in range(i+1,len(gray_images)):

        if i != j and((i,j) or (j,i) not in comp_list):
            # here we get only unique comparisons i.e comparing image 1 to image 2 and image 2 to image 1 are same.
            # we eliminate comparing image 2 and image 1 as image 1 and 2 are already compared.

            prev_frame = gray_images[i].copy()
            next_frame = gray_images[j].copy()

            # we are taking file names along with image files
            prev_frame_name = files[i]
            next_frame_name = files[j]

            # getting the contour areas that are calcuated in hand before comparison. Helps to reduce computation time

            min_contour_area = (min_contour_areas_list[i] + min_contour_areas_list[j]) / 2

            score,res_cnts,thresh = compare_frames_change_detection(prev_frame, next_frame, min_contour_area= min_contour_area)

            scores.append(score)


            count = 0
            if score == 0 and min_contour_areas_list[i] == min_contour_areas_list[j]:

                count += 1
                if count>0:
                    print('Duplicate Image found---  Image{} and Image{} are same'.format(files[i],files[j]))

                    # Duplicate Image names are appended to a list
                    if prev_frame_name not in duplicate_images_list:
                        duplicate_images_list.append(prev_frame_name)
```

Below is the detailed explanation of the loop shown in the image above.

Outer for loop is used to take the first image into consideration. The inner loop takes all the images other than the image taken in the outer loop. Here we name the first image as prev_frame along with the file name. Then we take the minimum contour areas we calculated beforehand. We take the average of two values. If the two images are the same, then the two images will have the same contour area and the score from the function compare_frames_change_detection will be zero. If the two images are different the score will be greater than 0. The more the two images are different the higher the score will be. We increase the value of the variable called count by 1 and then the prev_frame_name is appended into the duplicate_images_list.

If there are any duplicate images in the directory all the file names are in the list, and we remove the duplicate images in the directory.

```
if len(duplicate_images_list)>0:

    for i in range(len(duplicate_images_list)):

        os.remove(duplicate_images_list[i])
```