```
# Import necessary libraries
import os
import gc
import cv2
import torch
import imageio
import numpy as np
import matplotlib.pyplot as plt

from PIL import Image
from aot_tracker import _palette
from SegTracker import SegTracker
from scipy.ndimage import binary_dilation
from model_args import aot_args,sam_args,segtracker_argsge import binary_dilation
import gc

import joblib
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier


class SAM_Tracker_Pipeline:
   def __init__(self):
     # Dictionary to collect various metrics and results
     self.collect_dict = {
        "coords": None,
        "coordMaxThresholding": None,
        "coordMaxThresholding_maxVal": None,
        "coordMaxThresholding_minVal": None,
        "cumulativeSumThresholding": None,
        "cumulativeSumThresholding_yCoords": None,
      "bounceValidator": None
  }

# Method to save the segmentation mask predictions
def save_prediction(self, pred_mask, output_dir, file_name):
    save_mask = Image.fromarray(pred_mask.astype(np.uint8))
    save_mask = save_mask.convert(mode='P')
    save_mask.putpalette(_palette)
    save_mask.save(os.path.join(output_dir, file_name))

# Method to colorize the mask for visualization
def colorize_mask(self, pred_mask):
    save_mask = Image.fromarray(pred_mask.astype(np.uint8))
    save_mask = save_mask.convert(mode='P')
    save_mask.putpalette(_palette)
    save_mask = save_mask.convert(mode='RGB')
    return np.array(save_mask)

# Method to overlay the mask on the original image
def draw_mask(self, img, mask, alpha=0.7, id_countour=False):
    img_mask = np.zeros_like(img)
    img_mask = img
    if id_countour:
        # very slow ~ 1s per image
        obj_ids = np.unique(mask)
        obj_ids = obj_ids[obj_ids!=0]

        for id in obj_ids:
            # Overlay color on  binary mask
            if id <= 255:
                color = _palette[id*3:id*3+3]
                else:
                    color = [0,0,0]
                foreground = img * (1-alpha) + np.ones_like(img) * alpha * np.array(color)
                binary_mask = (mask == id)

                # Compose image
                img_mask[binary_mask] = foreground[binary_mask]

                countours = binary_dilation(binary_mask,iterations=1) ^ binary_mask
                img_mask[countours, :] = 0
        else:
            binary_mask = (mask!=0)
            countours = binary_dilation(binary_mask,iterations=1) ^ binary_mask
            foreground = img*(1-alpha)+colorize_mask(mask)*alpha
            img_mask[binary_mask] = foreground[binary_mask]
            img_mask[countours,:] = 0

        return img_mask.astype(img.dtype)



    # Method to generate the segmented prediction mask for the trajectory
    def process_video(self, video_name, output_dir='./PredMasks', prmpt_point=np.array([[0, 0]]), prmpt_labels=np.array([1]), save_in_dir=True):
        io_args = {
            'input_video': video_name,
            'output_mask_dir': output_dir # save pred masks
            # 'output_video': f'./assets/{video_name}_seg.mp4', # mask+frame visualization, mp4 or avi, else the same as input video
            # 'output_gif': f'./assets/{video_name}_seg.gif', # mask visualization
        }

        segtracker_args = {
```

```
    'sam_gap': 49, # the interval to run sam to segment new objects
    'min_area': 200, # minimal mask area to add a new mask as a new object
    'max_obj_num': 255, # maximal object number to track in a video
    'min_new_obj_iou': 0.8, # the area of a new object in the background should > 80%
}

# source video to segment
cap = cv2.VideoCapture(io_args['input_video'])
fps = cap.get(cv2.CAP_PROP_FPS)
# output masks
output_dir = io_args['output_mask_dir']
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
pred_list = []
masked_pred_list = []

torch.cuda.empty_cache()
gc.collect()
sam_gap = segtracker_args['sam_gap']
frame_idx = 0
segtracker = SegTracker(segtracker_args, sam_args, aot_args)
segtracker.restart_tracker()

with torch.cuda.amp.autocast():
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        if frame_idx == 0:
                pred_mask, _ = segtracker.seg_acc_click(frame, prmpt_point, prmpt_labels, multimask=True)
                torch.cuda.empty_cache()
                gc.collect()
                segtracker.add_reference(frame, pred_mask)
            elif (frame_idx % sam_gap) == 0:
                seg_mask, _ = segtracker.seg_acc_click(frame, prmpt_point, prmpt_labels, multimask=True)
                torch.cuda.empty_cache()
                gc.collect()
                track_mask = segtracker.track(frame)
                new_obj_mask = segtracker.find_new_objs(track_mask, seg_mask)
                if np.sum(new_obj_mask > 0) > frame.shape[0] * frame.shape[1] * 0.4:
                    new_obj_mask = np.zeros_like(new_obj_mask)
                if save_in_dir: self.save_prediction(new_obj_mask, output_dir, str(frame_idx) + '_new.png')
                pred_mask = track_mask + new_obj_mask
                segtracker.add_reference(frame, pred_mask)
            else:
                pred_mask = segtracker.track(frame, update_memory=True)
            torch.cuda.empty_cache()
            gc.collect()

            if save_in_dir: self.save_prediction(pred_mask, output_dir, str(frame_idx) + '.png')
            pred_list.append(pred_mask)

            print("processed frame {}, obj_num {}".format(frame_idx, segtracker.get_obj_num()), end='\r')
            frame_idx += 1
        cap.release()
        print('\nfinished')
```

```
    # Additional code for visualization if needed
    # masked_pred_list.append(masked_frame)
    # plt.imshow(masked_frame)
    # plt.show()

    # Return the list of predicted masks
    return pred_list

# Method to calculate the mean x and y coordinates of the mask
def calculate_xmean_ymean(self, mask_list):
    # Initialize an array to store [xmean, ymean] pairs
    xy_mean_array = []

    # Iterate over each masked image
    for mask_image in mask_list:
        # Extract the coordinates of the masked pixels
        y_coords, x_coords = np.where(mask_image > 0)

        # Calculate the mean x-coordinate and mean y-coordinate
        xmean = np.mean(x_coords)
        ymean = np.mean(y_coords)

        # Append [xmean, ymean] pair to the array
        xy_mean_array.append([xmean, ymean])

    # Convert the array to a NumPy array for consistency
    xy_mean_array = np.array(xy_mean_array)

    # Return the array of [xmean, ymean] pairs
    return xy_mean_array
```

```
#Method to get the maximum and minimum y-values of the trajectory
def trajectory_movements(self, trajectory_array):
    coords = np.array(trajectory_array)
    self.collect_dict['coords'] = coords

    max_value = np.max(coords[:, 0])
    min_value = np.min(coords[:, 0])

    return max_value, min_value

# Method to compute the difference between max and min y-values
def compute_coordMaxThresholding(self, pred_list, thresh=39):
    xy_mean_array = self.calculate_xmean_ymean(pred_list)
    # print(pred_list)
    max_value, min_value = self.trajectory_movements(xy_mean_array)
    self.collect_dict['coordMaxThresholding_maxVal'] = max_value
    self.collect_dict['coordMaxThresholding_minVal'] = min_value
    diff = max_value - min_value
    self.collect_dict['coordMaxThresholding'] = diff

    if diff >= thresh: return 1, diff
    else: return 0, diff

# Method to compute the cumulative sum thresholding
def find_min_y_position_and_index(self, trajectory_array):
    y_coords = trajectory_array[:, 1]
    min_index = np.argmin(y_coords)

    min_y = y_coords[min_index]
    return min_y, min_index, y_coords
```

```
  def calculate_max_distance(self, array, min_index):
    distances = np.abs(array[min_index:] - array[min_index])
    max_distance = np.max(distances)
    return max_distance

  def compute_cumulativeSumThresholding(self, pred_list, thresh=15):
    xy_mean_array = self.calculate_xmean_ymean(pred_list)
    min_value, min_index, y_coords = self.find_min_y_position_and_index(xy_mean_array)
    max_distance = self.calculate_max_distance(y_coords, min_index)
    self.collect_dict['cumulativeSumThresholding'] = max_distance
    self.collect_dict['cumulativeSumThresholding_yCoords'] = y_coords

    if max_distance > thresh: return 1
    else: return 0



if __name__ == '__main__':
  # Initializations
  rf_file_path = "/content/drive/MyDrive/BounceValidator/random_forest_model.pkl"
  loaded_rf_classifier = joblib.load(rf_file_path)

  vid_path = '/content/video_NV_63_3297_large.mp4'
  video_name = vid_path.split('/')[-1].split('.mp4')[0]

  # Testing the Pipeline
  pipeline = SAM_Tracker_Pipeline()
  pred_list = pipeline.process_video(vid_path, output_dir='./PredMasks',
                                     prmpt_point=np.array([[52, 109]]),
                                     prmpt_labels=np.array([1]),
                                     save_in_dir=False)
```

```
# Check for a validated bounce
x = pipeline.calculate_xmean_ymean(pred_list)
# Replace NaN values with a specific value (e.g., -9999)
# Reshape the data to a 1D array with 18 elements
x = x.flatten()
if len(x) < 18:
  x = np.pad(x, (0, 18-len(x)), mode='constant', constant_values=np.nan)

# Replace NaN values with a specific value (e.g., -9999)
nan_replacement_value = -9999
x = np.nan_to_num(x, nan=nan_replacement_value)

# Make predictions using the loaded Random Forest Classifier
loaded_rf_predictions = loaded_rf_classifier.predict(x[:18].reshape(1, -1))
pipeline.collect_dict['bounceValidator'] = loaded_rf_predictions[0]

res_coordMax, score = pipeline.compute_coordMaxThresholding(pred_list, thresh=39)
res_CumulativeSum = pipeline.compute_cumulativeSumThresholding(pred_list, thresh=39)

print(f'Result for Coord Max Thresholding is {res_coordMax}, with score: {score}')
print(f'Result for Cumulative Sum Thresholding is {res_CumulativeSum}')
```