```python
import os
import numpy as np
import json
from PIL import Image, ImageDraw

# Added by me
import pdb
import matplotlib.pyplot as plt
# Only used to provide a standard data structure to speed up clustering
from unionfind import unionfind
# import heapq as hq
# from scipy.optimize import minimize

### Helper Functions ###
def visualize(I, bounding_boxes, disp=False):
    im = Image.fromarray(I)
    draw = ImageDraw.Draw(im)
    for box in bounding_boxes:
        y0, x0, y1, x1 = box
        draw.rectangle([x0, y0, x1, y1])
    if disp:
        im.show()
    return np.asarray(im)

def saveImage(I, saveLoc):
    im = Image.fromarray(I)
    im.save(saveLoc)

def findRed(I, disp=False):
    im = Image.fromarray(I)
    hsv = np.asarray(im.convert('HSV'))

    hue = (hsv[:,:,0] < 50)  + (hsv[:,:,0] > 240) > 0
    sat = (hsv[:,:,1] > 130)
    value = (hsv[:,:,2] > 130)

    mask = hue * sat * value
    if disp:
        im = Image.fromarray(mask * 255)
        # im.show()
        plt.figure()
        plt.imshow(np.array(im))
    return np.transpose(np.where(mask)), mask

def fitCircle(cluster):
    pixels = np.array(cluster) # Nx2 now
    # Use center of mass as the center, look at how much distance of points
    # from center fluctuates, if too high then reject
    cm = np.mean(pixels, axis=0)
    r = np.mean(np.linalg.norm(pixels - cm))

    # Also return the circle
    return np.std(np.linalg.norm(pixels - cm)), cm, r

def findEdges(mask, disp=False):
    edgeMask = mask.copy()
    # Classify labeled pixel as edge if it has <= 3 neighbors in
    # cardinal directions
    numNeighbors = edgeMask[1:-1,2:].astype(int) + edgeMask[1:-1,:-2].astype(int) + edgeMask[:-2,1:-1].astype(int) +
edgeMask[2:,1:-1].astype(int)
    newNeighbors = np.zeros(numNeighbors.shape)

    # rnd = 0
    while np.sum(np.abs(newNeighbors - numNeighbors)) > 0:
        # print('On iteration', rnd)
        numNeighbors = newNeighbors
        padded = np.zeros(np.shape(edgeMask))
        padded[1:-1,1:-1] = (numNeighbors == 4)
        # Fill in any pixel that is surrounded
        edgeMask += padded.astype(bool)
        # Repeat now that filled in holes
        newNeighbors = edgeMask[1:-1,2:].astype(int) + edgeMask[1:-1,:-2].astype(int) + edgeMask[:-2,1:-1].astype(int) +
edgeMask[2:,1:-1].astype(int)
        # rnd += 1

    padded = np.ones(np.shape(edgeMask)) # Enable all pixels on perimeter edge
    padded[1:-1,1:-1] = (newNeighbors <= 3)
    edgeMask = padded * edgeMask

    if disp:
        plt.figure()
        plt.imshow(edgeMask)

    return np.transpose(np.where(edgeMask)), edgeMask

# Sliding window clustering where thresh dictates square window size
# mask should give
def clusterPixels(pixels, mask, thresh):

    pixelMap = {tuple(pixels[i]) : i for i in range(len(pixels))}

    u = unionfind(len(pixels))
    for i in range(mask.shape[0] - thresh):
        for j in range(mask.shape[1] - thresh):
            group = np.transpose(np.where(mask[i:i+thresh, j:j+thresh]))
            for k in range(len(group)):
                for l in range(k):
                    try:
                        u.unite(pixelMap[tuple(np.array([i,j]) + group[k])], pixelMap[tuple(np.array([i,j]) + group[l])])
                    except:
                        pdb.set_trace()
    groups = u.groups()

    clusters = [[pixels[i,:] for i in group] for group in groups]
    return clusters

# Implements the shared portion of the modified match filter and backboard approach
# and then specify with argument which approach to use for the last stage
# of processing
def initialPart(I, THRESH, MINSIZE, MAXSIZE, CIRCLETHRESH, ACCEPT, disp=False, BACKBOARD=False):
    bounding_boxes = []

    redPixels, mask = findRed(I, disp)

    # Find edges first
    pixels, edgeMask = findEdges(mask, disp)

    # List of lists where each inner list is one cluster of red pixels
    clusters = clusterPixels(pixels, edgeMask, THRESH)

    # Delete/ignore clusters which are too small
    clusters = [cluster for cluster in clusters if len(cluster) > MINSIZE and len(cluster) < MAXSIZE]

    clusterDraw = np.copy(I)
    for i, cluster in enumerate(clusters):
        color = np.zeros(3)
        color[i % 3] = 255
        for pixel in cluster:
            clusterDraw[pixel[0],pixel[1],:] = color

    im = Image.fromarray(clusterDraw)
    costs = []
    circles = []
    for cluster in clusters:
        pixels = np.array(cluster) # Nx2 now
        # Use center of mass as the center, look at how much distance of points
        # from center fluctuates, if too high then reject
        cm = np.mean(pixels, axis=0)
        radii = np.linalg.norm(pixels - cm, axis=1)
        r = np.mean(radii)
        cost = np.std(radii) / r # Fractional uncertainty/error

        costs.append(cost)

        if cost < CIRCLETHRESH:
            y, x = cm
            circles.append([x, y, r])
            draw = ImageDraw.Draw(im)
            draw.point([(x-2, y), (x-1, y), (x, y), (x+1, y), (x+2, y), (x, y-2), (x, y-1), (x, y+1), (x, y+2)])
            draw.ellipse((x-r, y-r, x+r, y+r))

    if disp:
        plt.figure()
        plt.imshow(np.array(im))
        # im.show()

    # print('Costs', costs)

    kernels, kernelCircles = loadTemplates()

    if BACKBOARD:
        print('Running Backboard')
        bounding_boxes = blackBack(I, mask, circles, ACCEPT[0], ACCEPT[1], False)
        # responsesList, candidatesList  = blackBack(I, mask, circles, kernels, kernelCircles, ACCEPT)
    else:
        print('Running Modified Match Filtering')
        responsesList, candidatesList = fitTemplates(I, circles, kernels, kernelCircles, ACCEPT)

        for i, responses in enumerate(responsesList):
            candidates = candidatesList[i]
            choice = np.argmax(responses)
            if responses[choice] > ACCEPT:
                bounding_boxes.append(candidates[choice])

    for box in bounding_boxes:
        y0, x0, y1, x1 = box
        draw.rectangle([x0, y0, x1, y1])
    if disp:
        plt.figure()
        plt.imshow(np.array(im))
        # im.show()

    return bounding_boxes

# Used only by the backboard approach, focuses on identifying the traffic
# light by the black backing
def blackBack(I, redFound, circles, threshBlack, threshJoined, disp=False):
    bounding_boxes = []

    for i, circle in enumerate(circles):
        x, y, r = circle
        d = 2*r

        # y0, x0, y1, x1
        bounds = [max(0, int(y-d)), max(0, int(x-1.3*d)), min(I.shape[0]-1, int(y+5*d)), min(I.shape[1]-1, int(x+1.3*d))]

        if disp:
```

```
        im = Image.fromarray(I)
        draw = ImageDraw.Draw(im)
        draw.point([(x-2, y), (x-1, y), (x, y), (x+1, y), (x+2, y), (x, y-2), (x, y-1), (x, y+1), (x, y+2)])
        draw.ellipse((x-r, y-r, x+r, y+r))
        draw.rectangle([bounds[1], bounds[0], bounds[3], bounds[2]])

    ROI = I[bounds[0]:bounds[2], bounds[1]:bounds[3]]

    redMask = redFound[bounds[0]:bounds[2], bounds[1]:bounds[3]]

    RED = 100
    GREEN = 100
    BLUE = 100
    # CUTOFF = 100
    # SAT = 100
    # VAL = 100

    # grayscale = ROI[:,:,0] * 0.2989 + ROI[:,:,1] * 0.5870 + ROI[:,:,2] * 0.1140

    # hsv = np.array(Image.fromarray(ROI).convert('HSV'))

    # mask = grayscale < 100
    mask = (ROI[:,:,0] < RED) * (ROI[:,:,1] < GREEN) * (ROI[:,:,2] < BLUE)
    # mask = (hsv[:,:,1] < SAT) * (hsv[:,:,2] < VAL)

    mask += redMask

    pixels, edgeMask = findEdges(mask, disp)

    # ymin, ymax, xmin, xmax
    defaults = [max(0, int(y-d)), min(I.shape[0]-1, int(y+4*d)), max(0, int(x-d)), min(I.shape[1]-1, int(x+d))]

    n = len(pixels)
    ALPHA = 0.1 # Fraction to trim
    if n:
        ySort = np.sort(pixels[:,0])
        xSort = np.sort(pixels[:,1])

        ymin = ySort[int(ALPHA * n)]
        ymax = ySort[n-1-int(n * ALPHA)]

        xmin = xSort[int(ALPHA * n)]
        xmax = xSort[n-1-int(ALPHA * n)]

        # ymin, xmin = np.min(pixels, axis=0)
        # ymax, xmax = np.max(pixels, axis=0)

        ymin += bounds[0]
        ymax += bounds[0]+1
        xmin += bounds[1]
        xmax += bounds[1]+1
    else:
        ymin = defaults[0]
        ymax = defaults[1]
        xmin = defaults[2]
        xmax = defaults[3]

    # If cannot find a reasonable side revert to defaults
    if ymin > y:
        ymin = defaults[0]
    if ymax < y:
        ymax = defaults[1]
    if xmin > x and xmax < x:
        xmin = defaults[2]
        xmax = defaults[3]
    elif xmin > x or xmax < x:
        if xmin > x:
            xmin = int(x - (xmax - x))
        if xmax < x:
            xmax = int(x + (x - xmin))

        # Look at fraction of black pixels in this region
        cropped = I[ymin:ymax, xmin:xmax]

        # cropped = np.array(Image.fromarray(cropped).convert('HSV'))

        # fracBlack = np.sum(grayscale < CUTOFF) / (cropped.shape[0] * cropped.shape[1])

        # fracJoined = fracBlack + np.sum(redFound[ymin:ymax, xmin:xmax]) / (cropped.shape[0] * cropped.shape[1])

        fracBlack = np.sum((cropped[:,:,0] < RED) * (cropped[:,:,1] < GREEN) * (cropped[:,:,2] < BLUE)) / (cropped.shape[0] *
cropped.shape[1])

        fracJoined = np.sum((cropped[:,:,0] < RED) * (cropped[:,:,1] < GREEN) * (cropped[:,:,2] < BLUE) + redFound[ymin:ymax,
xmin:xmax]) / (cropped.shape[0] * cropped.shape[1])

        # fracBlack = np.sum((cropped[:,:,1] < SAT) * (cropped[:,:,2] < VAL)) / (cropped.shape[0] * cropped.shape[1])

        # fracJoined = fracBlack + np.sum(redFound[ymin:ymax, xmin:xmax]) / (cropped.shape[0] * cropped.shape[1])

        if disp:
            plt.figure()
            plt.imshow(mask)
            draw.rectangle([xmin, ymin, xmax, ymax])

        if fracBlack > threshBlack and fracJoined > threshJoined:
            # Convert from np.int64 to int
            bounding_boxes.append([int(ymin), int(xmin), int(ymax), int(xmax)])

    if disp:
        plt.figure()
        plt.imshow(np.array(im))

    return bounding_boxes

# Fit the templates using the identified red circle for the modified match filter
def fitTemplates(I, circles, kernels, kernelCircles, thresh, disp=False):
    responsesList = []
    candidatesList = []

    for i, circle in enumerate(circles):
        x, y, r = circle

        responses = []
        candidates = []
        for j, orig in enumerate(kernels):
            # Resize the kernel to match the size of the circle
            xK, yK, rK = kernelCircles[j]
            scale = r / rK
            kernel = np.array(Image.fromarray(orig).resize((int(scale * orig.shape[1]), int(scale * orig.shape[0]))))

            xK = int(scale * xK)
            yK = int(scale * yK)
            start = (int(y - yK), int(x - xK))

            # Need to account for partially occluded traffic light
            tl_row = max(start[0], 0)
            tl_col = max(start[1], 0)
            br_row = min(start[0]+kernel.shape[0], I.shape[0])
            br_col = min(start[1]+kernel.shape[1], I.shape[1])

            if start[0] < 0:
                kernel = kernel[-start[0]:,:,:]
```

```python
            elif start[0] + kernel.shape[0] > I.shape[0]:
                kernel = kernel[:-(start[0] + kernel.shape[0] - I.shape[0]), :, :]
            if start[1] < 0:
                kernel = kernel[:,-start[1]:,:]
            elif start[1] + kernel.shape[1] > I.shape[1]:
                kernel = kernel[:, :-(start[1] + kernel.shape[1] - I.shape[1]), :]

            imPatch = I[tl_row:br_row, tl_col:br_col, :]

            K = np.ndarray.flatten(kernel).astype(np.float)
            patch = np.ndarray.flatten(imPatch).astype(np.float)

            K /= np.linalg.norm(K)
            patch /= np.linalg.norm(patch)

            try:
                response = np.dot(K, patch)
            except:
                pdb.set_trace()

            # if response > thresh:
            #     bounding_boxes.append([tl_row,tl_col,br_row,br_col])
            #     break
            responses.append(response)
            candidates.append([tl_row, tl_col, br_row, br_col])
        responsesList.append(responses)
        candidatesList.append(candidates)

    return responsesList, candidatesList

# Load the templates to be used by the modified match filter
def loadTemplates():
    kernels = []
    kernelCircles = []
    I = np.array(Image.open('data/RedLights2011_Medium/RL-010.jpg'))
    kernels.append(I[15:84, 124:173, :])
    # x, y, r
    kernelCircles.append([24, 13, 7])

    kernels.append(I[27:92, 321:349, :])
    kernelCircles.append([13, 13, 7])

    return kernels, kernelCircles

def detect_red_light(I):
    '''
    This function takes a numpy array <I> and returns a list <bounding_boxes>.
    The list <bounding_boxes> should have one element for each red light in the
    image. Each element of <bounding_boxes> should itself be a list, containing
    four integers that specify a bounding box: the row and column index of the
    top left corner and the row and column index of the bottom right corner (in
    that order). See the code below for an example.

    Note that PIL loads images in RGB order, so:
    I[:,:,0] is the red channel
    I[:,:,1] is the green channel
    I[:,:,2] is the blue channel
    '''

    # bounding_boxes = [] # This should be a list of lists, each of length 4. See format example below.

    # Setting to False enables the modified match filter to run
    BACKBOARD = False

    if BACKBOARD:
        # Backboard approach
        bounding_boxes = initialPart(I, 3, 7, 150, 0.4, [0.6, 0.75], True, BACKBOARD)
    else:
        # Modified matched filtering
        bounding_boxes = initialPart(I, 3, 7, 150, 0.4, 0.85, True, BACKBOARD)

    for i in range(len(bounding_boxes)):
        assert len(bounding_boxes[i]) == 4

    return bounding_boxes


### Constants which control program flow ###

# controls whether to save labeled images
saveLabeled = True
# controls whether to try to predict on full data
fullPred = False

######

# set the path to the downloaded data:
data_path = 'data/RedLights2011_Medium'

# set a path for saving predictions:
preds_path = 'data/hw01_preds'
os.makedirs(preds_path,exist_ok=True) # create directory if needed

### Running Code for Questions 4 and 5 ###

# Worst Examples
worst_names = ['RL-017.jpg', 'RL-064.jpg', 'RL-149.jpg', 'RL-273.jpg', 'RL-244.jpg']

# Best Examples
best_names = ['RL-031.jpg', 'RL-139.jpg', 'RL-196.jpg', 'RL-248.jpg']

for i, group in enumerate([worst_names, best_names]):
    path = preds_path + '/'+ 'worst'*(1-i) + 'best'*i
    for j, name in enumerate(group):
        print('On image ' + name)

        # read image using PIL:
        I = Image.open(os.path.join(data_path,group[j]))

        # convert to numpy array:
        I = np.asarray(I)

        bounding_boxes = detect_red_light(I)

        labeled = visualize(I, bounding_boxes, False)

        if saveLabeled:
            os.makedirs(path,exist_ok=True) # create directory if needed
            saveImage(labeled, path + '/labeled' + name + '.jpg')

#######

if fullPred:
    # get sorted list of files:
    file_names = sorted(os.listdir(data_path))

    # remove any non-JPEG files:
    file_names = [f for f in file_names if '.jpg' in f]

    # In-sample
    # file_names = ['RL-001.jpg', 'RL-002.jpg', 'RL-003.jpg', 'RL-004.jpg', 'RL-005.jpg', 'RL-006.jpg', 'RL-007.jpg', 'RL-008.jpg', 'RL-
009.jpg', 'RL-010.jpg']
```

```
    # Out-sample
    # file_names = ['RL-325.jpg', 'RL-326.jpg', 'RL-327.jpg', 'RL-328.jpg', 'RL-329.jpg', 'RL-330.jpg', 'RL-331.jpg', 'RL-332.jpg', 'RL-
333.jpg', 'RL-334.jpg']

    preds = {}
    for i in range(len(file_names)):
        print('On image ' + str(i))
        # read image using PIL:
        I = Image.open(os.path.join(data_path,file_names[i]))

        # convert to numpy array:
        I = np.asarray(I)

        bounding_boxes = detect_red_light(I)

        preds[file_names[i]] = bounding_boxes

        labeled = visualize(I, bounding_boxes, False)

        if saveLabeled:
            saveImage(labeled, preds_path + '/labeled' + str(i) + '.jpg')

    # save preds (overwrites any previous predictions!)
    with open(os.path.join(preds_path,'preds.json'),'w') as f:
        json.dump(preds,f)



### Dead Code ###

# def findEdges(mask, disp=False):
#     edgeMask = mask.copy()
#     # Classify labeled pixel as edge if it has <= 3 neighbors in
#     # cardinal directions
#     numNeighbors = edgeMask[1:-1,2:].astype(int) + edgeMask[1:-1,:-2].astype(int) + edgeMask[:-2,1:-1].astype(int) +
edgeMask[2:,1:-1].astype(int)
#     # Now, pad back with zeros
#     padded = np.zeros(np.shape(edgeMask))
#     padded[1:-1,1:-1] = (numNeighbors == 4)
#     edgeMask += padded.astype(bool)
#     # Repeat now that filled in holes
#     numNeighbors = edgeMask[1:-1,2:].astype(int) + edgeMask[1:-1,:-2].astype(int) + edgeMask[:-2,1:-1].astype(int) +
edgeMask[2:,1:-1].astype(int)
#     padded = np.ones(np.shape(edgeMask)) # Enable all pixels on perimeter edge
#     padded[1:-1,1:-1] = (numNeighbors <= 3)
#     edgeMask = padded * edgeMask

#     if disp:
#         plt.imshow(edgeMask)

#     return np.transpose(np.where(edgeMask)), edgeMask


# # Do agglomerative clustering with a max distance threshold
# # pixels should be Nx2
# def clusterPixels(pixels, thresh):
#     edges = []
#     for i in range(len(pixels)):
#         for j in range(i):
#             d = np.linalg.norm(pixels[i,:] - pixels[j,:])
#             hq.heappush(edges, (d, (i,j)))
#     u = unionfind(len(pixels))

#     minD = 0
#     while minD < thresh:
#         minD, edge = hq.heappop(edges)
#         if not u.issame(edge[0], edge[1]):
```

```
#             u.unite(edge[0], edge[1])

#     groups = u.groups()

#     clusters = [[pixels[i,:] for i in group] for group in groups]
#     return clusters

# def templateApproach(I, kernel, THRESH):
#     # Pad the image by the dimensions of the kernel / 2 using the mean value
#     # in the image
#     padSize = int(kernel.shape[0] / 2), int(kernel.shape[1] / 2)
#     paddedIm = np.ones((I.shape[0] + 2 * padSize[0], I.shape[1] + 2 * padSize[1], 3))
#     for i in range(3):
#         paddedIm[:,:,i] *= np.mean(I[:,:,i])
#     paddedIm[padSize[0]:I.shape[0] + padSize[0], padSize[1]:I.shape[1] + padSize[1], :] = I
#     Kall = kernel.astype(np.float32)

#     passed = np.ones((paddedIm.shape[0] - kernel.shape[0], paddedIm.shape[1] - kernel.shape[1]))

#     for channel in range(3):
#         print('On channel ' + str(channel))
#         K = np.ndarray.flatten(Kall[:,:,channel])
#         K /= np.linalg.norm(K)

#         # Now, do sliding window
#         response = np.zeros((paddedIm.shape[0] - kernel.shape[0], paddedIm.shape[1] - kernel.shape[1]))
#         for i in range(response.shape[0]):
#             for j in range(response.shape[1]):
#                 imPatch = np.ndarray.flatten(paddedIm[i:i+kernel.shape[0],j:j+kernel.shape[1],channel]).astype(np.float32)
#                 imPatch /= np.linalg.norm(imPatch)
#                 response[i,j] = np.sum(imPatch * K)
#         passed *= (response > 0.9)

#         Image.fromarray(response * 255).show()

#     Image.fromarray(passed * 255).show()
#     pdb.set_trace()






# def blackBack(I, circles):
#     for i, circle in enumerate(circles):
#         x, y, r = circle
#         d = 2*r

#         # y0, x0, y1, x1
#         bounds = [max(0, int(y-2*d)), max(0, int(x-3*d)), min(I.shape[0]-1, int(y+6*d)), min(I.shape[1]-1, int(x+3*d))]
#         im = Image.fromarray(I)
#         draw = ImageDraw.Draw(im)
#         draw.point([(x-2, y), (x-1, y), (x, y), (x+1, y), (x+2, y), (x, y-2), (x, y-1), (x, y+1), (x, y+2)])
#         draw.ellipse((x-r, y-r, x+r, y+r))
#         draw.rectangle([bounds[1], bounds[0], bounds[3], bounds[2]])

#         ROI = I[bounds[0]:bounds[2], bounds[1]:bounds[3]]

#         circleROI = [x-bounds[1], y-bounds[0], r]

#         # Find black pixels in ROI
#         hsv = np.array(Image.fromarray(ROI).convert('HSV'))

#         vROI = hsv[:,:,2]

#         mask = (vROI < 130)
#         # im.show()
#         plt.imshow(mask)

#         initGuess = np.array([d, d, 3*d])

#         # result = minimize(boxObj, initGuess, args=(circleROI, vROI))

#         pdb.set_trace()


# # circleROI should be defined relative to ROI coordinates
# def boxObj(params, circleROI, vROI):
#     rw, ru, rd = params
#     x, y, r = circleROI
#     yl = max(0, int(y-ru))
#     xl = max(0, int(x-rw))
#     yr = min(vROI.shape[0], int(y+rd))
#     xr = min(vROI.shape[1], int(x+rw))

#     insidePixels = []
#     outsidePixels = []

#     # Left
#     insidePixels.extend(vROI[yl:yr, xl].tolist())
#     try:
#         outsidePixels.extend(vROI[yl:yr, xl-1].tolist())
#     except IndexError:
#         pass

#     # Right
#     insidePixels.extend(vROI[yl:yr, xr].tolist())
#     try:
#         outsidePixels.extend(vROI[yl:yr, xr+1].tolist())
#     except IndexError:
#         pass

#     # Top
#     insidePixels.extend(vROI[yl, xl:xr].tolist())
#     try:
#         outsidePixels.extend(vROI[yl-1, xl:xr].tolist())
#     except IndexError:
#         pass

#     # Bottom
#     insidePixels.extend(vROI[yr, xl:xr].tolist())
#     try:
#         outsidePixels.extend(vROI[yr+1, xl:xr].tolist())
#     except IndexError:
#         pass

#     pdb.set_trace()

#     return (np.mean(insidePixels) - np.mean(outsidePixels))**2






# '''
# BEGIN YOUR CODE
# '''

# '''
# As an example, here's code that generates between 1 and 5 random boxes
# of fixed size and returns the results in the proper format.
# '''
# box_height = 8
# box_width = 6

# num_boxes = np.random.randint(1,5)

# for i in range(num_boxes):
#     (n_rows,n_cols,n_channels) = np.shape(I)

#     tl_row = np.random.randint(n_rows - box_height)
#     tl_col = np.random.randint(n_cols - box_width)
#     br_row = tl_row + box_height
#     br_col = tl_col + box_width

#     bounding_boxes.append([tl_row,tl_col,br_row,br_col])

# '''
# END YOUR CODE
# '''


# costList = []
# paramList = []
# for cluster in clusters:
#     params, cost = fitEllipse(cluster, ALPHA)
#     x, y, r = params

#     paramList.append(params)
#     costList.append(cost)

#     draw = ImageDraw.Draw(im)
#     draw.ellipse((x-r, y-r, x+r, y+r))

# im.show()
# pdb.set_trace()


# def identifyRed(imList):
#     hues = []
#     sats = []
#     vals = []
#     for I in imList:
#         im = Image.fromarray(I)
#         hsv = np.asarray(im.convert('HSV'))
#         hues.extend(np.ndarray.flatten(hsv[:,:,0]).tolist())
#         sats.extend(np.ndarray.flatten(hsv[:,:,1]).tolist())
#         vals.extend(np.ndarray.flatten(hsv[:,:,2]).tolist())
#     fig, axes = plt.subplots(1, 3)
#     pdb.set_trace()
#     n = 10**5
#     inds = np.random.choice(range(len(hues)), n)
#     axes[0].hist(np.array(hues)[inds])
#     axes[1].hist(np.array(sats)[inds])
#     axes[2].hist(np.array(vals)[inds])
#     axes[0].set_title('Hue')
#     axes[1].set_title('Saturation')
#     axes[2].set_title('Value')

#     pdb.set_trace()

#     return fig, axes

# if False:
#     files = ['data/redSamples/redChunk' + str(i)  + '.jpg' for i in range(1,6)]
#     imList = [np.asarray(Image.open(file)) for file in files]
#     identifyRed(imList)
#     pdb.set_trace()
# def approach2(I, kernel, THRESH):
#     K = kernel.astype(np.float32)
#     K = np.ndarray.flatten(K)
#     K /= np.linalg.norm(K)
#     # Pad the image by the dimensions of the kernel / 2 using the mean value
#     # in the image
#     padSize = int(kernel.shape[0] / 2), int(kernel.shape[1] / 2)
#     paddedIm = np.ones((I.shape[0] + 2 * padSize[0], I.shape[1] + 2 * padSize[1], 3))
#     for i in range(3):
#         paddedIm[:,:,i] *= np.mean(I[:,:,i])
#     paddedIm[padSize[0]:I.shape[0] + padSize[0], padSize[1]:I.shape[1] + padSize[1], :] = I

#     # Now, do sliding window
#     response = np.zeros((paddedIm.shape[0] - kernel.shape[0], paddedIm.shape[1] - kernel.shape[1]))
#     for i in range(response.shape[0]):
#         for j in range(response.shape[1]):
#             imPatch = np.ndarray.flatten(paddedIm[i:i+kernel.shape[0],j:j+kernel.shape[1],:]).astype(np.float32)
#             imPatch /= np.linalg.norm(imPatch)
#             response[i,j] = np.sum(imPatch * K)
#     passed = (response > 0.9)
#     im = Image.fromarray(response * 255)
#     im.show()

#     masked = Image.fromarray(passed * 255)
#     masked.show()
#     pdb.set_trace()


# def fitEllipse(cluster, alpha=0):
#     pixels = np.array(cluster) # Nx2 now
#     # initial guess
#     cm = np.mean(pixels, axis=0)
#     r0 = np.mean(np.linalg.norm(pixels - cm))
#     x0 = np.array(cm.tolist() + [r0])

#     # Regularize to further encourage being close to initial guess
#     obj = lambda params: np.sum(np.square(np.linalg.norm(pixels - params[:2], axis=1) - params[2])) + alpha *
np.linalg.norm(params - x0)
#     result = minimize(obj, x0)
#     # Switch order from y,x to x,y
#     x = result.x
#     temp = x[1]
#     x[1] = x[0]
#     x[0] = temp

#     return x, result.fun


# # Use kernels to dictate ROI
# def blackBack(I, circles, kernels, kernelCircles, thresh, disp=False):
#     responsesList = []
#     candidatesList = []

#     for i, circle in enumerate(circles):
#         x, y, r = circle

#         responses = []
#         candidates = []
#         for j, orig in enumerate(kernels):
#             # Resize the kernel to match the size of the circle
#             xK, yK, rK = kernelCircles[j]
#             scale = r / rK
#             kernel = np.array(Image.fromarray(orig).resize((int(scale * orig.shape[1]), int(scale * orig.shape[0]))))
#             xK = scale * xK
#             yK = scale * yK
#             start = (int(y - yK), int(x - xK))
#           # Need to account for partially occluded traffic light
#           tl_row = max(start[0], 0)
#           tl_col = max(start[1], 0)
#           br_row = min(start[0]+kernel.shape[0], I.shape[0])
#           br_col = min(start[1]+kernel.shape[1], I.shape[1])

#           if start[0] < 0:
#               kernel = kernel[-start[0]:,:,:]
#           elif start[0] + kernel.shape[0] > I.shape[0]:
#               kernel = kernel[:, :-(start[0] + kernel.shape[0] - I.shape[0]), :]
#           if start[1] < 0:
#               kernel = kernel[:, -start[0]]
#           elif start[1] + kernel.shape[1] > I.shape[1]:
#               kernel = kernel[:, :-(start[1] + kernel.shape[1] - I.shape[1]), :]

#           imPatch = I[tl_row:br_row, tl_col:br_col, :]

#           # Find black pixels in ROI
#           # hsv = np.array(Image.fromarray(imPatch).convert('HSV'))
#           # vROI = hsv[:,:,2]
#           # mask = (vROI < 130)

#           mask = (imPatch[:,:,0] < 70) * (imPatch[:,:,1] < 70) * (imPatch[:,:,2] < 70)

#           if disp:
#               plt.figure()
#               plt.imshow(mask)

#               plt.figure()
#           # Look at fraction of pixels in patch considered black
#           response = np.sum(mask) / np.prod(mask.shape)

#           pdb.set_trace()

#           responses.append(response)
#           candidates.append([tl_row, tl_col, br_row, br_col])

#       responsesList.append(responses)
#       candidatesList.append(candidates)
#   return responsesList, candidatesList
```