PROBLEM STATEMENT:

Given any picture containing people standing behind each other, identify if two people are standing too close.

STEPS:

1. Detect the object in the image as person.
2. Find the Euclidean Distance between two persons.
3. Find the Minimum/Shortest Distance between persons and display it.

WORKFLOW/LOGIC:

OpenCV has in-built methods to detect Pedestrian/Person Detection in an Image.

OpenCV uses pre-trained **H**istogram Of **O**riented **G**radients (HOG) and Linear **S**upport **V**ector **M**achine (SVM) model.

The use of **N**on-**M**axima **S**uppression Algorithm (NMS) is to take multiple, overlapping boundary boxes and reduce them to only a single boundary box.

This helps reduce the number of false-positives reported by the final object-detector.

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--images", required=True, help="path to images directory")
args = vars(ap.parse_args())

# initialize the HOG descriptor/person detector
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
```

Command-line argument **–images** which is the path to directory that contains the list of images we are going to perform Pedestrian/Person Detection.

To initialise person detection first make initialisation of HOG Descriptor using **hog=cv2.HOGDescriptor ().**

To set SVM to be pre-trained People Detector obtained by the function

cv2.HOGDescriptor_getDefaultPeopleDetector () function.

```
for imagePath in imagePaths:
        # load the image and resize it to (1) reduce detection time
        # and (2) improve detection accuracy
        image = cv2.imread(imagePath)
        image = imutils.resize(image, width=min(400, image.shape[1]))
        origi = image.copy()
```

Start looping over the –images path and resizing images for maximum width of 400 pixels.

Resizing Image improves overall accuracy of our person detection.

```
# detect people in the image
(rects, weights) = hog.detectMultiScale(image, winStride=(4, 4),
        padding=(8, 8), scale=1.05)
```

Detecting persons in images will be handled by function **detectMultiScale** with scale=1.05 and sliding window step size of (4,4) pixels in both x and y respectively and this function returns a 2-tuple rects or boundary box (x,y)-coordinates of each person in image.

```
# draw the original bounding boxes
for (x, y, w, h) in rects:
        cv2.rectangle(origi, (x, y), (x + w, y + h), (0, 0, 255), 2)
```

These lines draw our initial bounding boxes over the image.

```
# apply non-maxima suppression to the bounding boxes using a
# fairly large overlap threshold to try to maintain overlapping
# boxes that are still people
rects = np.array([[x, y, x + w, y + h] for (x, y, w, h) in rects])
pick = non_max_suppression(rects, probs=None, overlapThresh=0.65)

# draw the final bounding boxes
for (xA, yA, xB, yB) in pick:
        cv2.rectangle(image, (xA, yA), (xB, yB), (0, 255, 0), 2)

# show some information on the number of bounding boxes
filename = imagePath[imagePath.rfind("/") + 1:]
print("[INFO] {}: {} original boxes, {} after suppression".format(
        filename, len(rects), len(pick)))

# show the output images
cv2.imshow("After NMs",image)
cv2.waitKey(0)
```

After applying NMS we get final boundary boxes. The final lines of code gives output result in Window.

USAGE:

In Command Prompt (CMD) change the directory to the file directory.

$python distance_persons.py –image images

This command will display an output in the screen.

MEASURING DISTANCE BETWEEN OBJECTS (PERSONS)

First, we need to find distance between objects to find the shortest distance to find who are too close to each other and display it.

The reference object should satisfy two properties.

1. We should know the dimensions of Reference Object. Normal Person has a width of 15 inches.
2. We must easily find reference object in image.

```
def midpoint(ptA, ptB):
        return ((ptA[0] + ptB[0]) * 0.5, (ptA[1] + ptB[1]) * 0.5)

# load the image, convert it to grayscale, and blur it slightly
images = cv2.imread(args["images"])
width=imutils.resize(images,width=min(16, images.shape[1]))
gray = cv2.cvtColor(images, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (7, 7), 0)

# perform edge detection, then perform a dilation + erosion to
# close gaps in between object edges
edged = cv2.Canny(gray, 50, 100)
edged = cv2.dilate(edged, None, iterations=1)
edged = cv2.erode(edged, None, iterations=1)
```

First cv2 will read the image using **cv2.imread()** and maximum width of 16 inches. The image is then converted to a grayscale and then blur it using a Gaussian Filter with a 7*7 kernel.

Once image is blurred we apply Canny edge detector to detect edges in the image. Dilation (**dilate**) and Erosion (**erode**) is performed to close any gaps in the edge map.

```
# find contours in the edge map
cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)

# sort the contours from left-to-right and, then initialize the
# distance colors and reference object
(cnts, _) = contours.sort_contours(cnts)
colors = ((0, 0, 255), (240, 0, 159), (0, 165, 255), (255, 255, 0),
        (255, 0, 255))
refObj = None
```

cv2.findContours detects the outline of the objects in edge map and **sort_contours** will sort the contours from left-right.

We then initialise colours to draw distances along with refObj.

```
# loop over the contours individually
for c in cnts:
        # if the contour is not sufficiently large, ignore it
        if cv2.contourArea(c) < 100:
                continue

        # compute the rotated bounding box of the contour
        box = cv2.minAreaRect(c)
        box = cv2.cv.BoxPoints(box) if imutils.is_cv2() else cv2.boxPoints(box)
        box = np.array(box, dtype="int")

        # order the points in the contour such that they appear
        # in top-left, top-right, bottom-right, and bottom-left
        # order, then draw the outline of the rotated bounding
        # box
        box = perspective.order_points(box)

        # compute the center of the bounding box
        cX = np.average(box[:, 0])
        cY = np.average(box[:, 1])
```

Loop-over the contours (cnts) and call to **order_points** rearranges the bounding box of (x,y) coordinates.

cX and cY compute the centre (x,y) of the boundary box.

```
 if refObj is None:
          # unpack the ordered bounding box, then compute the
          # midpoint between the top-left and top-right points,
          # followed by the midpoint between the top-right and
          # bottom-right
          (tl, tr, br, bl) = box
          (tlblX, tlblY) = midpoint(tl, bl)
          (trbrX, trbrY) = midpoint(tr, br)

          # compute the Euclidean distance between the midpoints,
          # then construct the reference object
          D = dist.euclidean((tlblX, tlblY), (trbrX, trbrY))
          refObj = (box, (cX, cY), D / 15)
          continue
```

When refObj is None we need to initialise it and then find the corresponding mid-points.

```
# loop over the original points
for ((xA, yA), (xB, yB), color) in zip(refCoords, objCoords, colors):
          # draw circles corresponding to the current points and
          # connect them with a line
          cv2.circle(orig, (int(xA), int(yA)), 5, color, -1)
          cv2.circle(orig, (int(xB), int(yB)), 5, color, -1)
          cv2.line(orig, (int(xA), int(yA)), (int(xB), int(yB)),
                    color, 2)

          # compute the Euclidean distance between the coordinates,
          # and then convert the distance in pixels to distance in
          # units
          D = dist.euclidean((xA, yA), (xB, yB)) / refObj[2]
          (mX, mY) = midpoint((xA, yA), (xB, yB))
          print("Eucledian Distance",D)
          cv2.putText(orig, "{:.1f}in".format(D), (int(mX), int(mY - 10)),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.55, color, 2)
          cv2.imshow("Images",orig)
          cv2.waitKey(0)
          mD=3
          if(D<mD):
                    mD=D
                    print("Minimum",mD)
                    continue
```

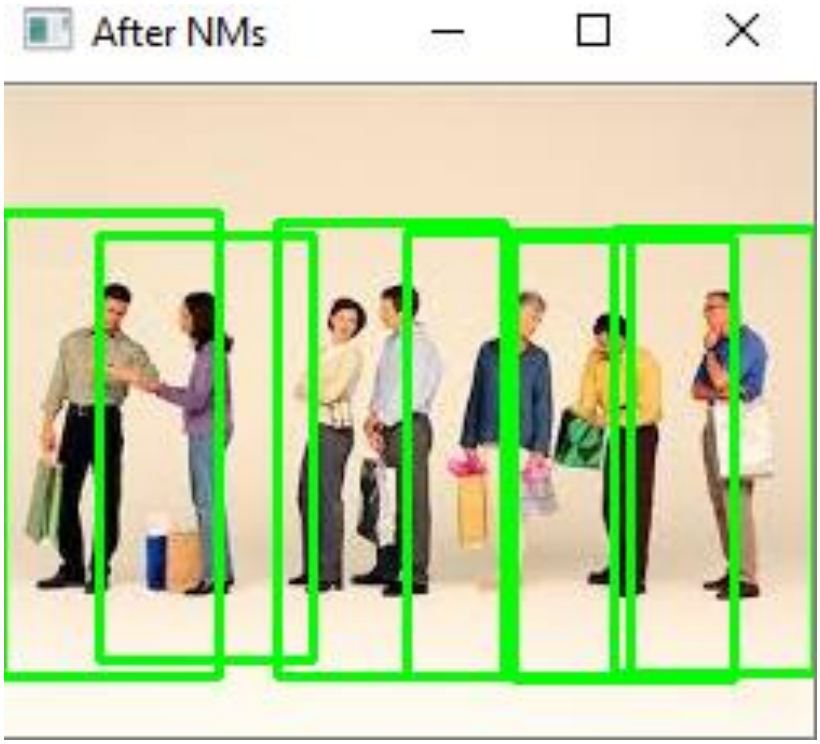
Then start looping over the points. Then compute the Euclidean Distance between Reference Object to Object_of_Interest

After Finding Euclidean Distance find the Minimum/Shortest Euclidean Distance between Persons and print it in an output.

OUTPUT:

To detect if the object is a Person perform the following command line.

$python distance_persons.py –image images



After Performing NMS showing the output result as persons detected.

To calculate the distance between persons use the command.

$python distance_persons.py –image images\final.jpg –method “right-to-left”

And if the distance is minimum than normal distance then it will display it as Minimum.