Social Distancing Detector

Ori Zur
Northwestern University

August 28, 2020

Abstract

The following is an in-depth overview of a social distancing detector program coded using Python and OpenCV. The project was created as a means to answer the question of how well are people following social distancing guidelines in outdoor urban environments. The program takes in a video as input, detects people using YOLOv3, calculates the distance between each pair of people, and indicates if a person is violating social distancing guidelines, or standing less than six feet away from another person. User inputted mouse coordinates and OpenCV's perspective transform functions are used to both create the region of interest for detection and calculate the number of pixels that make up the six-foot distance in the video being used. This program was created as part of the SAGE project, a cyberinfrastructure for weather and urban sensors that utilizes AI at the edge, allowing for near real-time analysis and data collection.

1 Introduction

For the past six months, the world has been enduring a historic pandemic due to the COVID-19 virus. As society attempts to adjust to the new lifestyle of mask wearing, virtual education, and working from home, one phrase that is constantly mentioned is "social distancing guidelines." Social distancing is the action of keeping a distance of at least six feet from others in order to reduce the spread of the Coronavirus disease. In public indoor settings, many businesses have enacted new policies to help people properly follow these guidelines such as drastically reducing a building's maximum occupancy and posting signage on the floors to help control and direct the flow of indoor traffic. However, when it comes to outdoor urban environments, people are largely left on their own to determine how to actively keep a safe distance from others. This more difficult circumstance, coupled with the fact that there exists a percentage of the population that is choosing to ignore these guidelines entirely, raises the following important question: what percentage of people are properly following social distancing guidelines in outdoor urban environments?

As a means to answer this question, a social distancing detector program was created using Python and OpenCV. The program takes in a video of pedestrians from an outdoor surveillance camera as input, and analyzes each frame of the video by detecting the people in the frame, calculating the distance between each pair of people, and indicating if any person is standing less than six feet from someone else. This program was created as part of the SAGE project at the Argonne National Laboratory. SAGE is a cyberinfrastructure of weather and urban sensors that utilizes "AI at the edge," a system in which computers are embedded within sensors so that data can be analyzed in near-real time. The following

is a detailed outline of the inner-workings of this program and how to run it.

2 Code Components

There are two main components to the program: the setup, which only occurs once in the beginning, and the operation, which is a loop that occurs once for each frame of the input video.

2.1 The Setup

When the program begins running, the first frame of the input video is shown to the user. The user then inputs six points with their mouse. The first four points make up a quadrilateral that approximates a rectangle on the ground plane, which will be referred to as the "region of interest" or ROI. The last two points approximate a six-foot distance on the ground. Ideally, there should be distance markers on the ground, objects of known dimensions, or other features in the image such as road lines and sidewalks to aid the user in accurately plotting the six points.

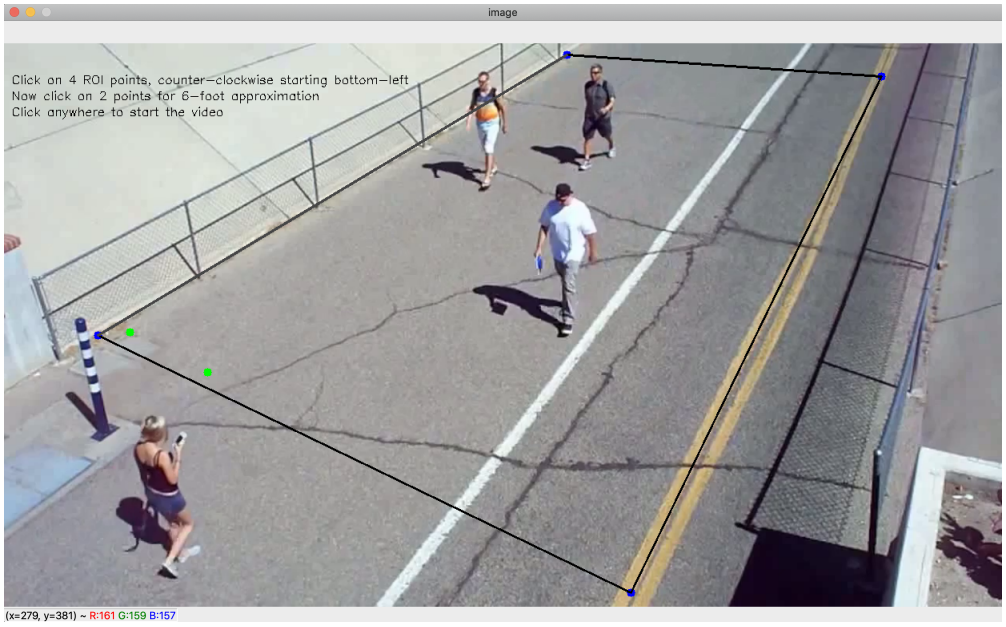

Figure 1: Sample of first frame mouse inputs. The four blue points represent the region of interest and the two green points represent the six-foot approximation (drawn by eye in this example). User instructions are given in the top-left corner.

The purpose of creating a region of interest with the first four mouse points is to solve the issue of camera distortion. Because the camera is filming from an angle, the conversion rate between physical distance on the ground and pixel distance in the image is not constant. In other words, the number of pixels that make up a six-foot distance on the ground

changes depending on the distance from the camera. The solution to this problem is to use the OpenCV function "cv2.getPerspectiveTransform." The function takes as arguments the coordinates of the first four mouse points as the "source" and the dimensions of the frame as the "destination" and outputs a 3x3 transformation matrix. This matrix is then used as a kernel to change the perspective of the last two mouse points using the function "cv2.perspectiveTransform," giving the two points new (x, y) coordinates. The distance formula is then used to calculate the distance between these two new coordinates, and the result, known as the minimum safe distance, is a float value that represents the number of pixels that make up a six-foot distance on the ground plane. Because the two points are warped using the transformation matrix before calculating the distance between them, this pixel value is constant for any six-foot distance, so long as the two points clicked in the image are on the ground plane.

As a proof of concept, a small-scale experiment was performed using LEGO figures as human substitutes. Four figures were placed on a piece of graph paper and a rectangle was drawn around them. On the sides of the rectangle, equidistant tick marks were drawn. Images of the figures were taken from angles approximating those of video surveillance footage. This angle caused the tick marks on the side of the rectangle to distort, meaning that the pixel distance between each tick mark decreased as the distance from the camera increased. The pictures were run through the same perspective transformation function described above in order to create the transformation matrix. The drawn rectangle was used as a guide to input the mouse points. The matrix was then used to run the entire image through the function "cv2.warpPerspective," which warps the entire image rather than specific points. The result of this program was an image in which the LEGO figures looked misshapen and distorted, but the pixel distances between each tick mark on the sides of the rectangle appeared to be equal. This result indicated that a transformation matrix could be used to make the conversion rate between the physical distance on the ground to pixel distance in the image constant. This method will allow for more accurate distance calculations and detections of social distancing violations.

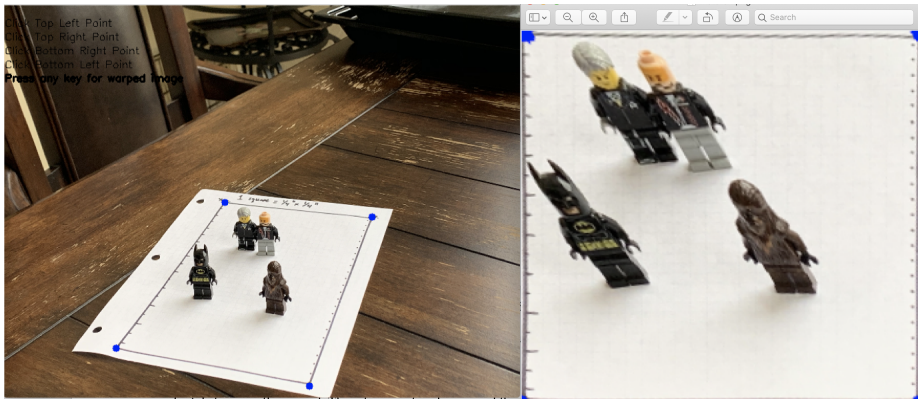

Figure 2: LEGO small-scale experiment. Left is original image, right is transformed image.

2.2 The Operation

The second component of the code is the operation, which is a while loop that occurs once for each frame of the input video until either the video ends or the user ends the program

manually. The first step in the operation loop is person detection, which is accomplished using a real-time object detection program called You Only Look Once, or YOLOv3. YOLO recognizes a wide variety of objects including people, animals, furniture, vehicles, and many other common items. This specific program included a filter so that only human detections were kept. In addition, one of the arguments passed into the program by the user is a confidence value between zero and one that filters out weak and uncertain detections by YOLO. The default value for this argument is 0.5 but it can be changed using the "--confidence" flag when running the program. Once detection occurs, the results are stored as a list of bounding boxes, or rectangles whose coordinates surround the person being detected. The final part of this step is to apply an algorithm called Non-Maximum Suppression, which removes all extraneous boxes to ensure that there is only one box for each person. This algorithm takes in a threshold argument between zero and one that is defaulted to 0.3 but can be changed by the user using the "--threshold" flag when running the program.

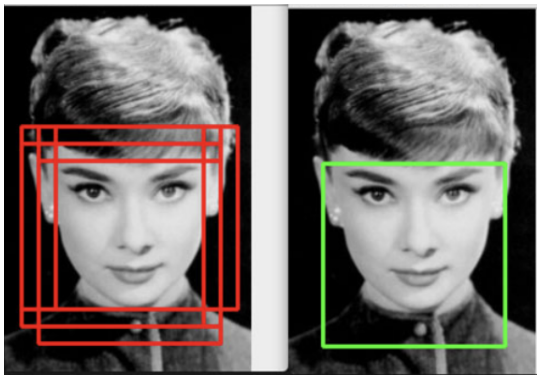

Figure 3: Example that shows the effects of the Non-Maximum Suppression algorithm.

The next step in the operation loop is to assign unique object IDs to each detected person. In order to do so, the program uses a class called the Centroid Tracker. This class assigns unique object IDs to each bounding box and keeps a register of people that have been detected in previous frames. The class contains an update function which is called every frame. The function takes in a list of bounding boxes that were outputted by YOLO in the current frame and compares these input boxes with the boxes currently in the register. The Euclidian distance is calculated between each pair of input boxes and registered boxes in order to associate each input box with an object that has already been detected. The assumption is that the input box that is closest to an already registered object must be the same object, and so that object's location is updated once the association has been made.

If there are more input boxes than boxes in the register, it means that a new detection has occurred, and so the program assigns it a new ID and the object is added to the register. If there are fewer input boxes than boxes in the register, it means that an object from the register was not detected in the most recent frame, and so it is marked "disappeared." If an object is marked as disappeared for too many consecutive frames, that object is removed from the register. The number of consecutive disappeared frames is defaulted to 10 but can be altered by the user using the "--disappeared" flag when running the program. After the update function is called, the results are organized into a list of tuples where the first element is the object ID and the second element is the coordinates and dimensions of that

object's bounding box. Using this method, people detected in the video can be assigned unique object IDs, allowing the user to keep track of specific individuals across multiple frames, rather than analyzing each frame independently.

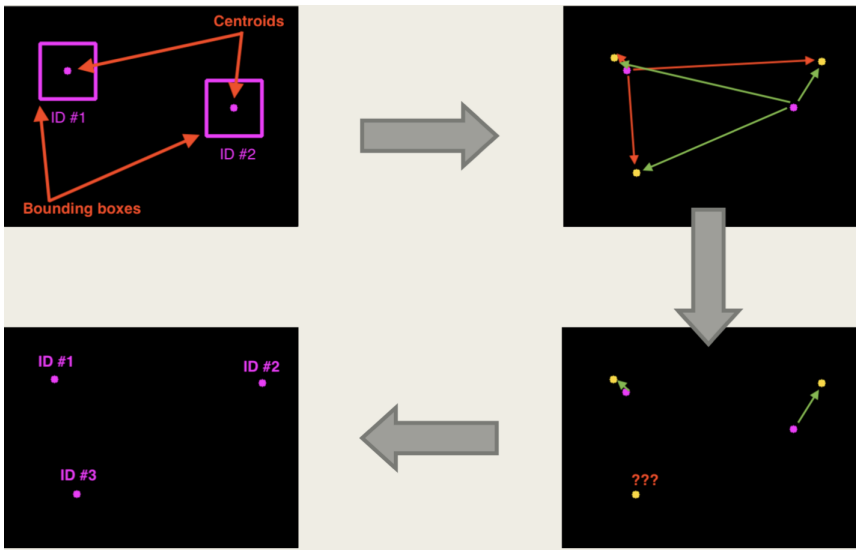

Figure 4: Diagram showing the steps of the update function of the centroid tracking class. Pink dots are boxes currently in the register and yellow dots are new detections being passed into the function. In this case, there are more yellow dots than pink dots, so a new ID is given to the dot that wasn't associated with a box already in the register.

The third step of the operation loop is calculating the distance between each pair of detected people and marking those who are standing less than six feet from another person. This part of the program employs the aid of a series of helper functions. The first helper function, called "transform_box_points," takes as input the list of object ID – bounding box tuples as well as the transformation matrix from the setup phase. It first calculates the coordinate of the bottom-center point of each bounding box, and then applies the transformation matrix to warp each of these bottom-center points. The function returns a list of tuples where the first element is the object ID and the second element is the new warped coordinate associated with that object's box.

The second helper function is called "violation_detection," and it takes as input the list of object ID – bounding box tuples, the list of object ID – warped coordinate tuples generated from the first helper function, and the minimum safe distance calculated in the setup phase. The function uses the warped coordinates list to calculate the distance between each pair of bounding boxes and compares the distances to the minimum safe distance. Each pair of bounding boxes is then tagged with a Boolean called "safe" which is true if the distance between the two boxes is greater than the minimum safe distance and false if the distance between the two boxes is less than the minimum safe distance. The function returns two outputs: the first is a list of lists where each inner list contains two bottom-point coordinates and the safe Boolean associated with them. The second output is also a list of lists, but here every inner list contains two bounding box coordinates and dimensions and the safe Boolean associated with them. Finally, the first output of this function is passed through

the third helper function, called "get_violation_count," which counts and returns the number of people who are currently violating social distancing guidelines and the number of people who are safe.

The final step of the operation loop is creating and displaying the outputs for the current frame. The first output is called the street view. It displays the video that was inputted by the user with the addition of red and green rectangles drawn over the bounding boxes of those violating social distancing guidelines and those who are safe respectively. Each person's unique object ID is also drawn above the rectangle. In the top-left corner of the window, the total number of people detected in the current frame, the number of violators, and the number of safe people is displayed. The second output, called the Bird's Eye View, is a window that represents the warped region of interest as a white rectangle. On the window, green and red circles are drawn using the coordinates of the warped bottom-center point of each bounding box. Red lines are also drawn between pairs of red circles to represent pairs of people that are too close together. Assuming that the four region of interest points are entered in the correct order, the circles will correspond with the people in the street view walking inside the region of interest.

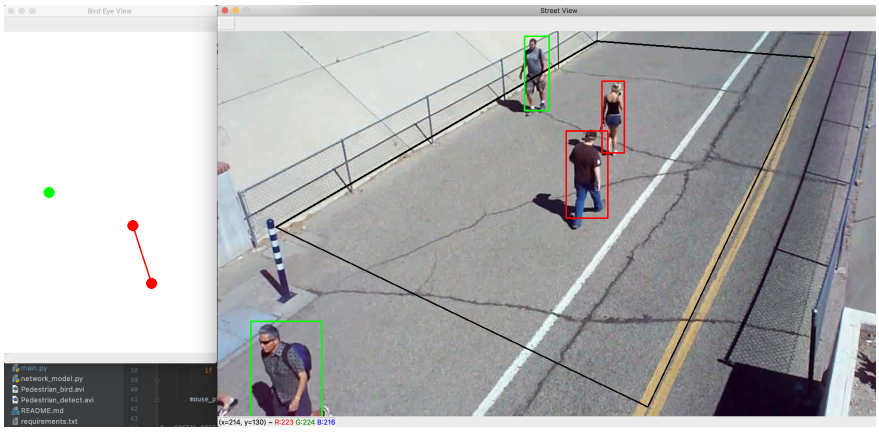

Figure 5: Left: Bird's eye view output. Right: Street view output
Note: Only those who are walking within the region of interest, represented by the black rectangle, appear in the bird's eye view output.

The final output is a text file that updates every X number of frames, where X defaults to twenty but can be adjusted by the user using the "--frames" flag when running the program. Each update adds the following information to the text file: current date and time, a list of object IDs and their current coordinates, number of safe pedestrians, number of violator pedestrians, and a social distancing ratio, which is a value from zero to one calculated by dividing the number of violators by the total number of pedestrians detected that frame. This output can be found as "out.txt" in the project folder as soon as the program stops running.

```
6 feet = 279 pixels.

Current time: 2020-08-21 14:58:28
Pedestrians Detected: 4
        Object ID: 1[362, 436]
        Object ID: 2[589, 365]
        Object ID: 3[901, 609]
        Object ID: 4[674, 762]
Safe Pedestrians: 0
Violator Pedestrians: 4
Social Distancing Ratio: 1.0

Current time: 2020-08-21 14:58:34
Pedestrians Detected: 4
        Object ID: 1[404, 512]
        Object ID: 2[588, 443]
        Object ID: 3[928, 692]
        Object ID: 4[665, 703]
Safe Pedestrians: 0
Violator Pedestrians: 4
Social Distancing Ratio: 1.0

Total Time Elapsed: 00:00:22.06
```

Figure 6: Sample screenshot of out.txt

The following is a flowchart that summarizes the steps of the code. The orange boxes represent the setup component and the green boxes represent the operation loop.

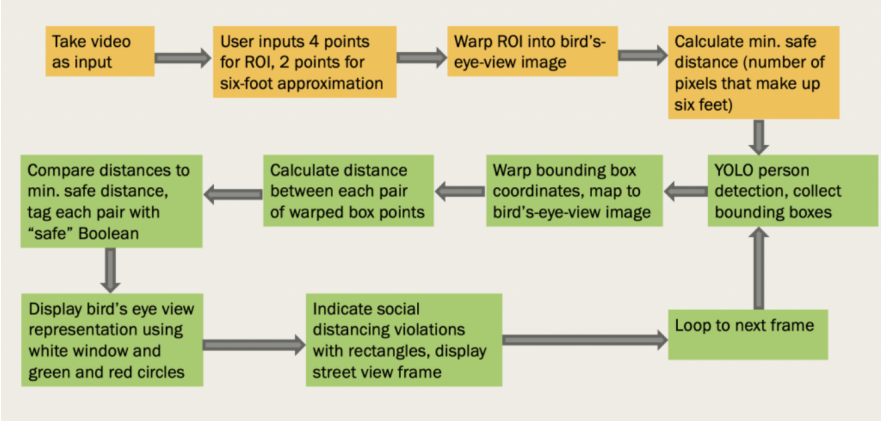

Figure 7: Flowchart of the different code components. Orange boxes represent setup steps and green boxes represent operation steps.

3 Program Usage

The python code for this program can be found here:
https://github.com/ozur1/SocialDistancingDetector

How to use the program:

1. Download the YOLO object detector: ./download_model.sh
2. Install all dependencies: pip install -r requirements.txt
3. Run the program:

(a) Using computer's webcam: python sdd.py

(b) Using a pre-recorded video: python sdd.py --input (path to video)

4. Other arguments that can be adjusted:

   (a) --confidence: float, confidence threshold for yolo detection (default=0.5)

   (b) --threshold: float, non-maximum suppression algorithm threshold (default=0.3)

   (c) --disappeared: int, number of consecutive frames that a registered object ID is not detected before it is de-registered (default=10)

   (d) --frames: int, number of frames between data outputs to text file (default=20)

5. Follow the instructions on the image that pops up to input the six mouse points for the region of interest and the six-foot approximation

4 Next Steps and Conclusion

This social distancing detector program serves as a means to answer the question of how well people are following social distancing guidelines in outdoor urban environments. This program successfully detects and calculates distances between people, however there are many different ways to organize the data that is collected. Currently, the data is being outputted onto a text file as the program runs. However, other similar methods could be done that would make the data easier to parse when it is being analyzed such as using JSON dumps. Additionally, the program currently doesn't save the actual distances between each pair of people; it only compares the distances with the minimum safe distance and tags the pair of bounding boxes with a true of false value. If the actual distance between the boxes was converted from pixels back to feet and saved alongside the safe Boolean, then this opens up many additional possibilities for data collection. An example of this would be to keep track of the average distance between each pair of people and plotting these values vs. time as a way to see how the pedestrian density of a specific area changes over time. One final addition that could be added to the program is to create multiple "zones" of safety. In other words, a pedestrian who is standing less than six feet from another person may be marked as "in danger," but a pedestrian who is standing between six and ten feet from another person would be marked as "low risk," instead of "safe." Overall, the information provided in this paper, along with the many comments posted throughout the code, could help a potential user alter or make additions to the code in order to output the desired data and organize it to suit his or her needs.

While social distancing related to the COVID-19 pandemic may have been the primary inspiration for this program, this project could also be used in a wide variety of other circumstances. For example, if a street camera detects that there is a large crowd of people waiting to cross the street, the program can signal the traffic light to change sooner. If a camera in an elevator lobby detects a large number of people waiting to use the elevators, the program can call multiple elevators to that floor instead of sending them one at a time. The concepts used in this program such as human detection and perspective transformation could be used to create a more general human proximity detector, and the many potential applications for this kind of program could serve a key role in the ever-growing field of computer vision, specifically regarding crowd size and density estimation.

5 Acknowledgments

Dr. Scott Collis, SAGE Instrumentation Chair
Dr. Nicola Ferrier, SAGE Deputy Director
Dr. Pete Beckman, SAGE Director
Dr. Rajesh Sankaran, SAGE Platform Developer
Dr. Jennifer Dunn, NAISE Director of Research

References

[1] https://www.cdc.gov/coronavirus/2019-ncov/prevent-getting-sick/social-distancing.html

[2] https://github.com/deepak112/Social-Distancing-AI/

[3] https://github.com/aqeelanwar/SocialDistancingAI

[4] https://www.pyimagesearch.com/2020/06/01/opencv-social-distancing-detector/

[5] https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/

[6] https://www.pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/

[7] https://www.pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/

[8] https://www.pyimagesearch.com/2014/08/25/4-point-opencv-getperspective-transform-example/