Fire Sensor Project

Capstone Final Report

By: Sameeha Boga, Arya Goyal, Daniel Fontaine, Eve Mooney, Daniela Salazar, Natalia Wilson

Advisor: Jose Angel Martinez Lorenzo

Table of Contents

Initials

ABSTRACT

The Fire Sensor project team has designed and built a device capable of detecting people during fires using RF and optical sensors. This dual sensor system aims to assist first responders by providing real-time data on the location and number of individuals trapped in burning buildings, particularly in dense urban environments. Emergency services will benefit from enhanced situational awareness and improved efficiency of rescue operations. The ability to quickly and accurately identify the position of individuals in hazardous environments can be critical in minimizing casualties.

The Fire Sensor system utilizes Texas Instruments mmWave cascading imaging radar and the Intel RealSense camera. Radars, which transmit electromagnetic waves and read the associated reflection, generate a point cloud that maps the presence and position of individuals in a room. The RealSense camera, when activated, scans the room and offers a real time view of the situation. Combining the radar and the RealSense modules allow for the system to generate a more precise image of a given room. Upon the integration of machine learning modules, trained by the team to specifically identify human figures, the final product can report to emergency services the number of individuals still in an area along with their location. This component eliminates the potential of false identifications or the potential for excluding human figures in the sheer amount of data produced.

A key goal of this project was to create an efficient fire detection and response system. Aimed to assist urban environments where high-rise buildings and complex layouts can make search and rescue operations challenging. Recent events in big American cities, such as Los Angeles, have especially demonstrated the need for smarter solutions to disaster in aging cities. High-rise buildings often feature multiple floors and enclosed areas, hindering traditional rescue operations. The Fire Sensor system aims to decrease these challenges by offering rescuers accurate real-time information over the location of residents. The project's portal, designed with an easy to use UI, facilitates this goal further.

To test and validate project efficacy, the team developed a fully operational prototype, outfitted with the previously mentioned sensors. Testing was conducted by simulating the hazy environment. The prototype was enclosed in a secure acrylic case. One side was replaced with a fiberboard partition, allowing for a stopgap and smoke funnel to be cut into the side. An HVAC pipe slowly pumped fog from a fog machine until a realistic degree of opacity was achieved. Team members were posed in front of the sensors as it gathered data, in a variety of poses and

distances with several props. Consistently, sensors supplied data through the smoke while the ML models reported the existence of humans in the room.

Through and through, the Fire Sensor system is clearly capable of identifying people and their locations in low visibility environments. While it could stand to benefit from a more comprehensive training data set and more efficient radar, the project produced a novel solution that stands to better the lives of many. As this system continues to be developed and improved it will become a reliable and helpful tool for first responders to improve the safety and efficiency during fires.

INTRODUCTION

In 2023 alone, there were 1,389,00 fires in the United States according to FEMA [3]. 3,670 people have died from fire incidents and 13,350 were injured [3]. Additionally, the financial loss due to the fires in 2023 was an estimated $23.2 billion. In emergency fire situations, it can be difficult for firefighters and first responders to navigate through the densely packed urban environments. The responders may have to find their ways through new apartment layouts, work their way through thick smoke which blocks visibility, and locate humans to save them from the fires.

To help alleviate the challenges firefighters face, our team decided to implement an efficient fire detection system that utilizes a dual sensor approach to locate individuals in fires. Our fire sensor aims to provide coordinates on the individuals trapped in a room regardless of visibility through a website. This way, firefighters can anticipate how many individuals are in a room and where. Using this information, firefighters can make informed decisions and respond immediately saving both lives and reducing property damage.

PROBLEM FORMULATION

The goal of the Fire Sensor Project is to create a real-time detection system that indicates to first responders where people are located in burning buildings. This system will be built from an Intel RealSense camera, a radar, and have an online portal. The devices will be integrated into buildings primarily in urban environments. The whole system will combine data from the RealSense and the modular radar to give real time information to first responders. Unlike other thermal imaging or GPS based systems that are usually ineffective in heavy smoke or in urban environments the dual sensor approach will allow for the design to work despite any challenges and harsh conditions.

The system will be set up to continuously take data from both the RealSense camera and the modular radar at real time then generate coordinate-based outputs that pinpoint where an individual is located. The information will then be transmitted to a cloud based platform and then be displayed on an user friendly interface. This user-friendly interface will be accessible to firefighters through any device like a phone, tablet, or computer. The dual sensor approach will ensure for less error to occur. Through both points of data the system can ensure what is being detected is a person and not an object. This will also detect a person despite any obstructions.

Overall, the Fire Sensor Project system will ensure to work through various obstructions and challenges. This dual sensor system will surpass other gps technologies on the market. Most current technologies are too expensive or don't detect through obstructions and smoke.

ANALYSIS OF MAJOR SYSTEMS

REALSENSE ROBOFLOW

Roboflow is a web-based platform for managing computer vision workflows, specifically for training and deploying object detection models. It was founded in 2020 and supports a wide range of model architectures, including YOLOv3 (You Only Look Once, version 3), a real-time object detection algorithm known for its speed and accuracy [4]. YOLOv3 operates by dividing an image into a grid and simultaneously predicting bounding boxes and class probabilities [1]. As a result it enables fast and efficient detection of multiple objects in a single forward pass.

Roboflow enables users to annotate image datasets, apply preprocessing and augmentation, and export them in YOLOv3-compatible formats, including PyTorch. The image below illustrates Roboflow's complete computer vision pipeline, which spans from dataset collection and labeling to model training and deployment (Fig 1) [2]. After processing the data, users can train models using established frameworks like YOLOv3 and manage them through custom uploads or pre-trained foundation models. Finally, trained models can be deployed across various environments, including cloud platforms, on-premise systems, or edge devices. This streamlined workflow supports the rapid development of accurate, production-ready vision applications.

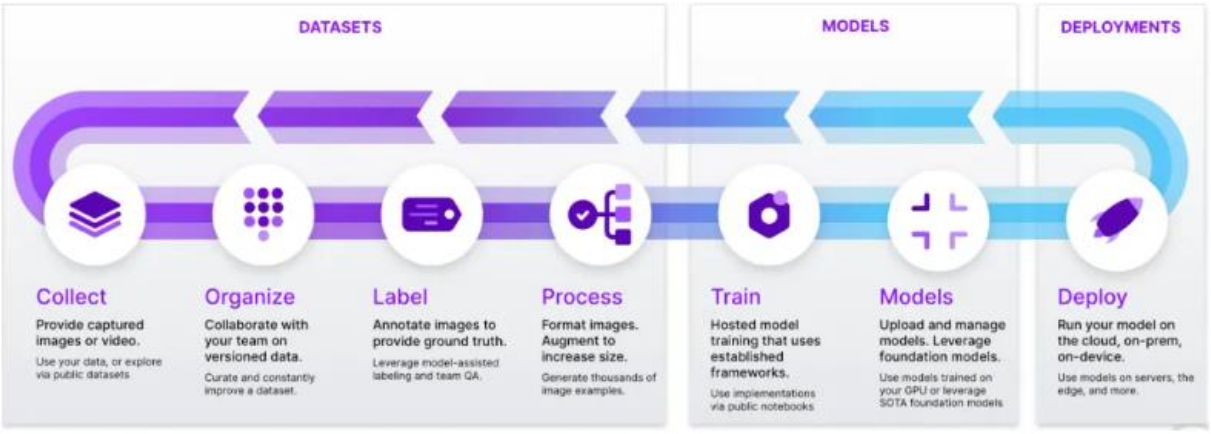

Fig. 1. Roboflow's complete computer vision pipeline. [2]

When paired with the Intel RealSense depth camera, Roboflow can be used to train a custom human detection model using both RGB and depth imagery to enhance detection robustness in dynamic environments. By automating dataset preparation and supporting model training and deployment, Roboflow streamlines the development of YOLOv3-based systems, making it well-suited for embedded vision applications that require accurate, low-latency detection, such as real-time human detection and tracking.

REALSENSE POINT CLOUD

In addition to the camera, the RealSense can also support the visualization of the point cloud. We used the RealSense camera to capture depth and color data captured to create a 3D representation of a scene. The depth camera measures the distance of objects from the sensor, while the color camera provides texture information. The point cloud is generated by mapping each pixel in the depth image to a 3D coordinate in space using the camera's intrinsic parameters.

The Python code initializes the RealSense camera pipeline and configures it to stream both depth and color data at 848x480 resolution and 60 FPS. It aligns the depth frames with the color frames to ensure accurate depth measurements corresponding to the correct color pixels. To enhance data quality, it applies spatial, temporal, and hole-filling filters to the depth frames. The script then retrieves the depth scale to convert raw depth values into meters. Next, it converts the depth and color frames into Open3D-compatible formats and constructs an RGBD image. Using the camera's intrinsic parameters, it generates a point cloud, which is then transformed and down-sampled to optimize visualization and storage. The processed 3D point coordinates are saved to a text file while the point cloud is displayed in an interactive Open3D visualizer. Additionally, the script prints the FPS for performance monitoring and ensures proper termination of the camera pipeline when the script ends. This process enables real-time 3D scene reconstruction, making it

valuable for applications like object scanning, robotics, and augmented reality. See Figures 2 and 3 below for examples of the 3D scene.

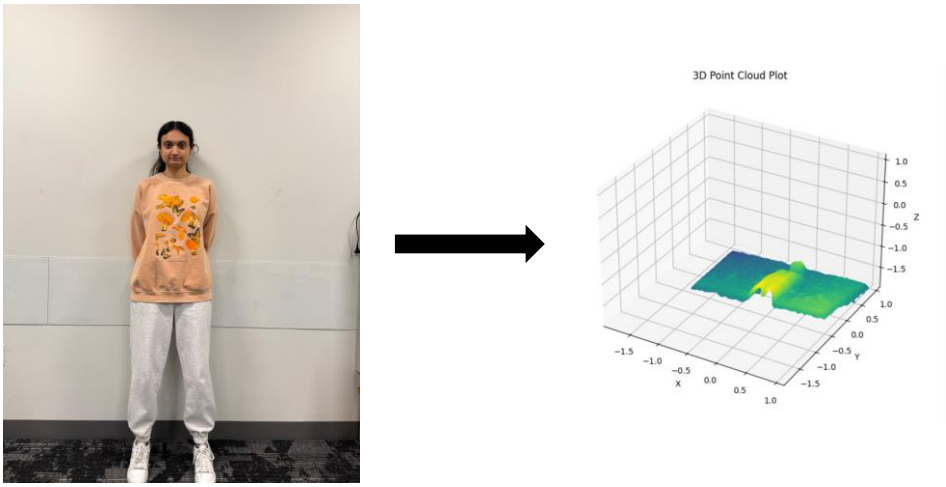

Fig. 2. Result of the RealSense Point Cloud on an Individual Standing

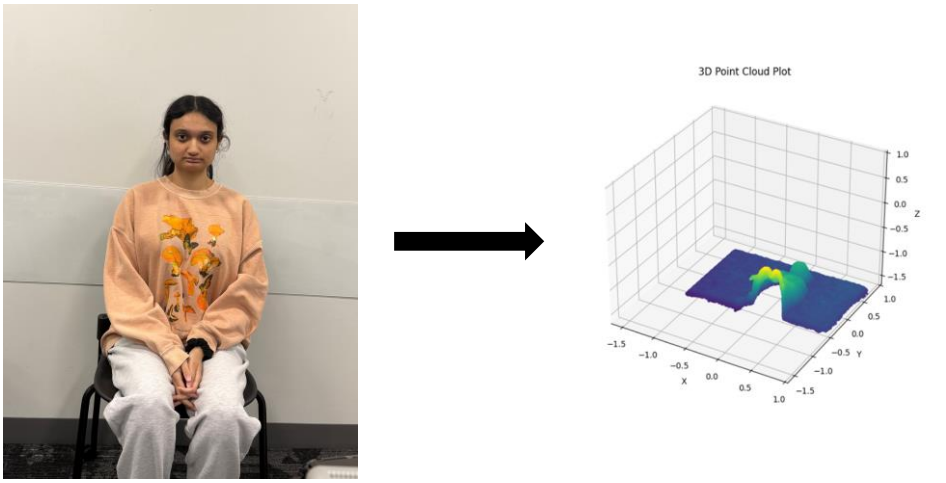

Fig. 3. Result of the RealSense Point Cloud on an Individual Sitting

RADAR DBSCAN

The Cascading radar uses electromagnetic pulses to scan the surrounding environment. We chose an electromagnetic radar due to the penetrating properties of EM waves. Electromagnetic waves are billions of times longer than that of visible light. Short wavelengths, like visible light or ultraviolet waves, are more prone to scattering. This can be seen in Rayleigh's scattering equation.

$$I = I_0 \times \frac{1 + cos\theta}{2} \times (\frac{\lambda}{\lambda^4}) \times N \times \sigma$$

However, this equation can be greatly simplified.

$$I = \frac{1}{\lambda^4}$$

This simplified equation shows I (the intensity of scattering), is inversely proportional to λ (the wavelength of the light). This same principle is what causes the sky to be a shade of blue. While the wave is not scattered by the small particles in the smoke, it will still be reflected by solid objects such as walls, chairs, and people. By timing how long it takes the wave to hit an object and come back to the radar, mmlabs can place a point at the calculated distance. By doing this multiple times the mmlabs is able to map thousands of points, creating a complete scan of the room. These individual scans are called iterations. To initialize and run the radar through mmlabs, we were provided with matlab code. Using this code, we can determine the number of iterations before each interpretation of the data, as well as some of the radar's settings.

```
'obj_RCS', 1, ... % RCS of the object [default = 1m^2]
'DSP_SNR_min', 15, ... % minimum SNR required by the imaging algorithum to detect the object [default = 10dB]
'Rx_VGA_gain', 30, ... % Rx VGA gain with all even values from 24 to 48 [default = 30dB]
'opt_priority', [10,1,10,1], ... % optimization preority of [range_max_obj range_res_obj velocity_max_obj velocity_
'radar_initialization', 0, ... % MM-wave front-end initialization using STUDIO APP
...                             % Set it to 1 whenever you modify the chirp profile
... %
... %
... %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
... %%%%%%%%%%%%%% Hardware and Software Settings %%%%%%%%%%%%%%
... %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'cascaded_mode', 0, ... % using 4-chip cascaded radar if 1; otherwise, set it to 0
'enable_real_time', 1, ... % enable_real_time==1 will force num_frames==0 such that infinite number of frames will
'cli_trigger_delay', 0.2, ... % interval between 'start_record' and 'stop_record' when enable_real_time==1
'server_IP', '0.0.0.0', ... % server IP address
'COM_NUM', 3, ... % COM port corresponding to 'XDS110 Class Application/User UART' when cascaded_mode==0
'raw_top_dir', 'c:\radar_data\demo', ... % top directory to dave raw .bin files
'bg_removal_server', 0, ... % apply background removal on imaging processing
'bg_bin_file_server', '*.bin', ... % binary file of the background measurement
'bg_removal_client', 0, ... % apply background removal on imaging processing
'bg_bin_file_client', '*.bin', ... % binary file of the background measurement
'server_save_multibins', 1,... % set to 1 if saving individual .bin file for each frame in the server when enable_r
'request_raw_adc_data', 0, ... % streaming the raw ADC data to the client through socket if request_raw_adc_data==1
'client_save_multibins', 1,... % set to 1 if saving individual .bin file for each frame in the client when enable_r
'num_frames', 0, ...% number of frames within one measurement; 0 means infinite
'frame_periodicity', 100, ... % frame periodicity in ms
'BPM', 0, ... % using binary phase modulation if 1
'NumMeas', 10, ... % total number of measurements: each measurement can have multiple frames; and each frame can ha
'window_flag', 1, ... % Hamming window for th                                          ADC data
'range_filtering', [0 0.2 200], ... % [whether_enable_rang_filter, range_min_to_keep, range_max_to_keep]
'server_visualization', 1, ... % 0: no visualization in the server end; 1: visulization without saving figures; 2:
'client_visualization', 0, ... % 0: no visualization in the client end; 1: visulization without saving figures; 2:
'client_figure2_window', [0.05 0.2 0.1 0.1], ... % window size of the figure 2
... %
... %
```

Fig. 4. Matlab initialization and settings code

After the predetermined amount of iterations, the radar will interpret all the scans before beginning another cycle of iterations. Each iteration is appended to the same list. This way, the radar can build a better picture of the environment over time. While this data is a rough map of the general environment of the room it is still useless to our first responders. For a clearer view the data has to be processed through a density based spatial clustering program.

```python
def plot_epsilon_neighborhoods(self):
    fig = plt.figure(figsize=(10, 7))
    ax = fig.add_subplot(121, projection='3d')
    ax.scatter(self.X[:, 0], self.X[:, 1], self.X[:, 2], alpha=0.7)
    self.plot_spheres(self.X[:5], self.eps, ax)
    ax.set_title(f"Epsilon Neighborhoods (eps={self.eps})")

    ax2 = fig.add_subplot(122, projection='3d')
    ax2.scatter(self.X[:, 0], self.X[:, 1], self.X[:, 2], alpha=0.7)
    ax2.scatter(self.X[0, 0], self.X[0, 1], self.X[0, 2], s=100, c='red')
    ax2.set_title(f"Core Point (min_pts={self.min_pts})")
    plt.show()

def euclidean_distance(self, point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))

def get_neighbors(self, point_idx):
    distances = [self.euclidean_distance(self.X[point_idx], other_point) for other_point in self.X]
    return [i for i, dist in enumerate(distances) if dist <= self.eps]

def find_core_points(self):
    core_points = []
    for i in range(len(self.X)):
        if len(self.get_neighbors(i)) >= self.min_pts:
            core_points.append(i)
    return core_points

def expand_cluster(self, labels, point_idx, neighbors, cluster_id):
    labels[point_idx] = cluster_id
    i = 0
    while i < len(neighbors):
        neighbor = neighbors[i]
        if labels[neighbor] == -1:  # Noise becomes border point
            labels[neighbor] = cluster_id
        elif labels[neighbor] == 0:  # Unvisited
            labels[neighbor] = cluster_id
            new_neighbors = self.get_neighbors(neighbor)
            if len(new_neighbors) >= self.min_pts:
                neighbors.extend(new_neighbors)
        i += 1
    return labels

def dbscan(self):
    labels = self.labels.copy()
    core_points = self.find_core_points()

    for point_idx in range(len(self.X)):
        if labels[point_idx] != 0:
            continue
        if point_idx in core_points:
            self.cluster_id += 1
            neighbors = self.get_neighbors(point_idx)
            labels = self.expand_cluster(labels, point_idx, neighbors, self.cluster_id)
        else:
            labels[point_idx] = -1  # Noise

    return labels
```

```python
def get_cluster_sizes(self):
    labels = self.dbscan()
    cluster_sizes = []
    for label in set(labels):
        if label != -1:  # Ignore noise
            cluster_sizes.append(labels.count(label))  # Count the number of points in this cluster
    return cluster_sizes

def get_average_distance_in_cluster(self):
    labels = self.dbscan()
    cluster_avg_distances = []

    # For each cluster, calculate the average distance between all pairs of points
    unique_labels = set(labels)
    for label in unique_labels:
        if label == -1:  # Ignore noise
            continue
        cluster_points = self.X[np.array(labels) == label]
        total_distance = 0
        count = 0
        for i in range(len(cluster_points)):
            for j in range(i + 1, len(cluster_points)):
                total_distance += self.euclidean_distance(cluster_points[i], cluster_points[j])
                count += 1
        if count > 0:
            cluster_avg_distances.append(total_distance / count)
        else:
            cluster_avg_distances.append(0)  # If no pairs, append 0

    return cluster_avg_distances

def plot_clusters(self):
    labels = self.dbscan()
    unique_labels = set(labels)
    colors = plt.cm.rainbow(np.linspace(0, 1, len(unique_labels)))

    fig = plt.figure(figsize=(10, 7))
    ax = fig.add_subplot(111, projection='3d')

    # Plot clusters
    for label, color in zip(unique_labels, colors):
        if label == -1:
            color = 'gray'  # Use gray for noise points
        class_member_mask = (np.array(labels) == label)
        xy = self.X[class_member_mask]
        ax.scatter(xy[:, 0], xy[:, 1], xy[:, 2], c=[color], alpha=0.7, label=f'Cluster {label}')
```

Fig. 5. Final DBScan code

Density based spatial clustering, or DBScan, is a type of program that will interpret the sea of points generated by the radar into separate objects known as clusters. All other points not included in the clusters will be discarded or added to cluster number -1. All points are still included in the graphs. However, they are now color coded to each cluster. This way, separate objects can be easily differentiated. Clusters are created using two values: the epsilon neighborhood radius, and the minimum points in that radius. If a point has the minimum number of points within its epsilon radius it becomes a core point. Each cluster is then formed around the core points. The cluster expands until they reach a discarded point, which will become a border point for that cluster. A core point within another core points neighborhood will combine their neighborhoods to create the final clusters. After the clusters are created, information about them is also calculated like the total number of points, and the average distance between points. These values are used to create and be interpreted by our binary classifier.

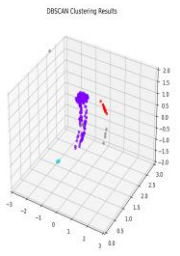

Fig. 6. DBScan testing results

To classify people using the radar, a binary random forest classifier was created through the SKlearn python library. The classifier is defined as binary because it either identifies the cluster as human or as other. The code feeds the clusters one by one to be interpreted by the algorithm. A random forest classifier is built on the idea of redundancy to interpret data. By combining multiple algorithms, or decision trees, to look at different aspects of a cluster, the algorithm can more accurately determine whether the cluster is human or not. Before the algorithms can be used, they need to be trained and saved so that they can be accessed instantly at any time. Using this code, we can create and save the random forest based on the values gathered through testing. By giving the algorithm values of human clusters, as well as values for whatever objects were present during the scan, the algorithm learns what is human as well as what is not. While the code was able to successfully identify people, more time and a larger data set can only improve the algorithm.

```
class RandomForestBinaryClassifier:
    def __init__(self, n_estimators=100, test_size=0.2, random_state=42):
        self.n_estimators = n_estimators
        self.test_size = test_size
        self.random_state = random_state
        self.model = RandomForestClassifier(n_estimators=self.n_estimators)

    def fit(self, X, y):
        # Convert X to numpy array if it's a list
        X = np.array(X)  # Ensure X is a numpy array

        # Ensure X is 2D, reshape if necessary
        if len(X.shape) == 1:  # If X is a 1D array
            X = X.reshape(-1, 1)  # Reshape X into 2D (n_samples, n_features)

        # Split data into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=self.test_size, random_state=self.random_state)

        # Convert labels to numerical values (0 for 'other', 1 for 'human')
        self.y_test = [1 if label == 'human' else 0 for label in y_test]

        # Train the model
        self.model.fit(X_train, [1 if label == 'human' else 0 for label in y_train])

        # Store test data for evaluation
        self.X_test = X_test

        # Store the original training data for comparison
        self.X_train = X_train

    def predict(self, X):
        # Convert X to numpy array if it's a list
        X = np.array(X)  # Ensure X is a numpy array

        # Ensure X is 2D, reshape if necessary
        if len(X.shape) == 1:  # If X is a 1D array
            X = X.reshape(-1, 1)  # Reshape X into 2D (n_samples, n_features)

        return self.model.predict(X)

    def evaluate(self, y_pred, y_true):
        # Evaluate the model
        accuracy = accuracy_score(y_true, y_pred)
        return accuracy

    def predict_labels(self, y_pred):
        # Convert predictions back to labels (0 for 'other', 1 for 'human')
        return ['human' if pred == 1 else 'other' for pred in y_pred]
```

```
def load_model_and_training_data(filename='numpoints.pkl'):
    """Load the trained RandomForest model and training data from a file."""
    data = joblib.load(filename)  # Load the model and training data (X_train)
    model = data['model']
    X_train = data['X_train']
    return model, X_train
def load_model(filename='avgdist.pkl'):
    """Load the trained RandomForest model and training data from a file."""
    data = joblib.load(filename)  # Load the model and training data (X_train)
    model = data['model']
    X_train = data['X_train']
    return model, X_train

# Example usage of loading the model and making predictions
if __name__ == "__main__":
    # Load the trained model and training data
    model, X_train = load_model_and_training_data('numpoints.pkl')

    # Sample data for prediction (new input)
    X_new = cluster_sizes  # Replace with actual data for prediction

    # Check if any value in X_new is within ±3 of any value in any row of X_train
    def check_within_tolerance(X_new, X_train, tolerance=3):
        """Check if any value in X_new is within ±tolerance of any value in X_train."""
        for row in X_train:
            for num in X_new:
                # Check if the difference between num and any value in row is within the tolerance
                if any(abs(num - x) <= tolerance for x in row):
                    return True
        return False

    # If any value in X_new is within ±3 of any value in X_train, return 1
    if check_within_tolerance(X_new, X_train):
        H=H+1
        #print("Prediction: 1")
    else:
        # Make a prediction using the model
        # print(f"Prediction: {y_pred}")
        H=H+0
if __name__ == "__main__":
    # Load the trained model and training data
    model, X_train = load_model('avgdist.pkl')

    # Sample data for prediction (new input)
    X_new = avg_distances  # Replace with actual data for prediction

    # Check if any value in X_new is within ±3 of any value in any row of X_train
    def check(X_new, X_train, tolerance=.05):
        """Check if any value in X_new is within ±tolerance of any value in X_train."""
        for row in X_train:
            for num in X_new:
                # Check if the difference between num and any value in row is within the tolerance
                if any(abs(num - x) <= tolerance for x in row):
                    return True
        return False

    # If any value in X_new is within ±3 of any value in X_train, return 1
    if check(X_new, X_train):
        H=H+1
        #print("Prediction: 1")
    else:
        # Make a prediction using the model
        #print(f"Prediction: {y_pred}")
        H=H+0
```

Fig. 7. Code for creating and calling the Random Forest classifier

At the end of the code, all of the data on the clusters the code has generated will be compared one by one to the AI datasets. If at least one cluster in the room is identified as a person, the code will place a bounding box around it, after which both the room status and the image will be uploaded. This process can be almost instantaneous depending on the amount of iterations between each processing. For testing purposes, the radar would run 10 iterations per processing. This would lead to the radar view of the room being 10 seconds behind and updating every 10 seconds. The room status updates twice as fast.

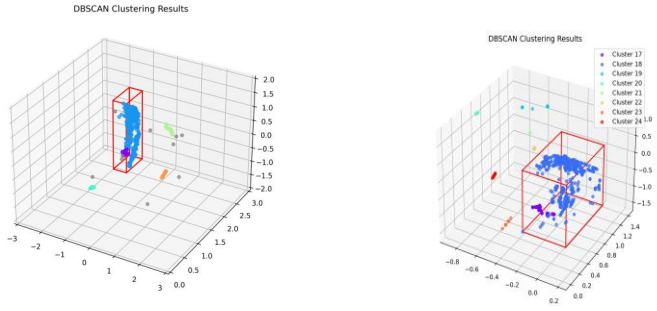

Fig. 8. Final DBScan results

DESIGN STRATEGIES AND APPROACH

REALSENSE VS WEB CAM

The Fire Sensor Project needed a device to see individuals in rooms that could also map the points in 3D versions. With this in mind we decided to use the Intel RealSense camera. The other option we considered was using a web camera. The choice between both cameras came down to the best features that assisted in completing this project.

The RealSense camera uses active infrared sensing and stereo vision to generate 3D depth maps while the webcam only sees 2D visuals. A very important feature we needed was to create 3D depth maps to ensure a person was being detected. As well as the RealSense camera could dictate how faraway individuals were. This feature assisted in the decision making as this was a main feature we needed for this project. Other features that made the RealSense camera a better option was the enhanced computer vision, ability to see through harsh conditions, and easy modular integration. RealSense has built-in SDKs for facial recognition and spatial awareness which allows enhanced computer vision. While the webcam would need extra software which could compromise the accuracy and speed. Another feature that was necessary was the ability to see through harsh conditions. This was necessary so that the camera could see through smoke and obstructed areas. The last feature that was necessary was the ability to be integrated with other software and hardware. The webcam would not allow for easy integration which would be a lot more difficult.

Due to all the features the RealSense was the better option for this project. The RealSense camera would allow for individuals to be seen through smoke and locate exactly where an individual is.

RADAR VS LIDAR

For mapping and surveying the environment through smoke, there were several different proposed methods. Ultrasonic and other sound based detection methods are capable of perfectly penetrating smoke without the risk of scattering. However, these methods are not capable of generating detailed scans of entire environments efficiently. To achieve similar results to other methods, upwards of 10 ultrasonic sensors, which became impractical. LIDAR, or Light Detection and Ranging sensors, had the opposite problem. While LIDAR is capable of producing detailed point clouds of rooms, the high wavelength nature of the technology made it more prone to scattering in the smoke. Electromagnetic waves provided a middle between these two options.

The second factor of the radar is speed. The radar needs to be able to get a complete scan of a room and determine whether anyone is trapped before first responders arrive. This can only be accomplished by taking as many scans of the room as possible. The cascading radar accomplishes this by effectively scanning the environment four times for each iteration. This is possible due to the fact that the cascading radar has one master radar and three slave radars. Each radar sends and receives pulses independently. All of the data is then interpreted and combined allowing scans of the environment to be incredibly fast.

Unfortunately, while testing the programs for the radar, hardware issues with the cascading radar constantly slowed us down. This severely hindered our ability to collect cloud data for our data sets. While we did collect 27 images of people, this isn't nearly enough for a good data set. The hardware problem continued through smoke testing. To resolve this problem we unfortunately had to switch from the cascading to the single radar. All the code works between both radars. However, while the cascading was emitting four signals at a time, the single radar can only emit and receive one. This means fewer points per iteration, or more time to scan an entire room. This means that the time taken to scan an entire room will likely decrease if the demos were run with a functioning cascading radar.

PARTS AND IMPLEMENTATION

ONLINE PORTAL AND UI

For the software part of our implementation, we designed an online portal for firefighters and first responders to view the state of the room. The portal displays both the camera feed and the point clouds from both the radar and the Realsense.

The portal, which we have implemented as a website, utilizes both python and Django for the backend. For the frontend, it utilizes HTML and javascript. Finally, the portal is hosted as a website on render.com, making it accessible to anyone with the link (Fire Sensor Server).

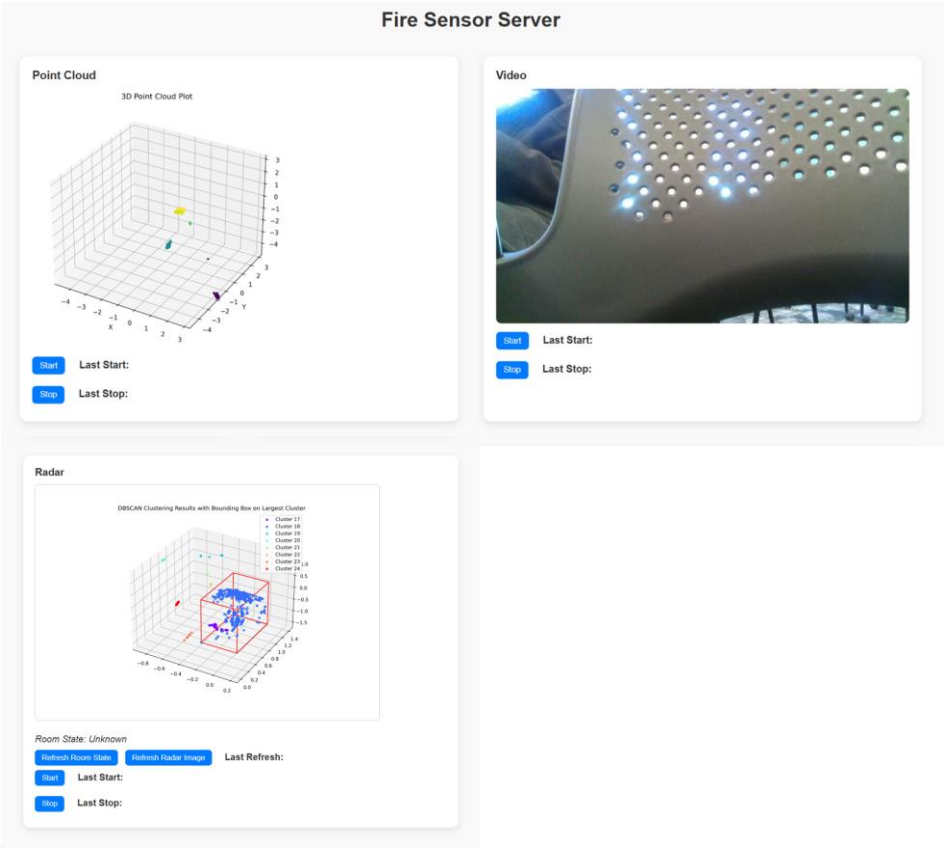

Fig. 9. Fire Sensor Website Components

The website displays the results of all three components of our prototype: the Realsense camera, the Realsense point cloud, and the radar point cloud. Each display consists of a button to start and stop data collection. This operates when the server and data collectors are running simultaneously.

The website was designed to be user friendly. It clearly displays the images that we are getting from the Realsense camera. Additionally, the website displays all three components making it easier to compare data and interpret the current state in the room.

The website utilizes a remote server, to receive and send information.

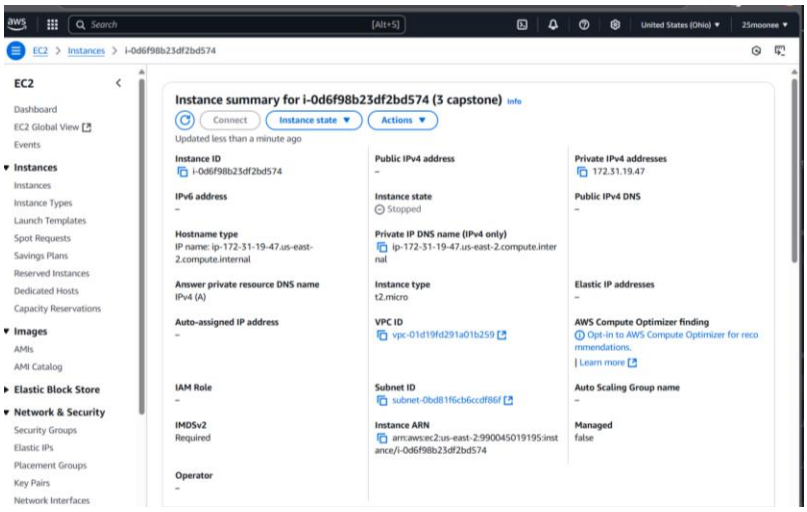

Fig. 10. Remote Server Instance

The instance for the remote server is an amazon AWS EC2. To implement this, we first created the instance using Ubuntu and connected the instance over SSH. We also added a flask server, to handle requests when running. it runs over HTTP.

The remote server receives information from the Realsense and the radar. Then, the website queries the information and displays them. The information is sent within the scripts that run to collect data.

Additionally, the server can handle the start and stop features. If the script for data collection is running, it will wait to start until the start button is pressed on the website. Then, it will collect data and continually send the updated results to the server. The website will re-query the server for new information every 3 seconds in order to display it on the page. This will continue until the stop button is pressed, ending the data collection and stopping the page from requerying for information.

The website gets different types of data. It receives a data file from the Realsense point cloud and parses it into a diagram. Additionally, it retrieves a video feed from the flask server that is connected to the website. Finally it retrieves and displays an image from the radar and also determines whether a room is clear. Multiple types of data can be displayed and updated on the website at the same time.

OUTSIDE ENCLOSURE DESIGN

When designing the outside enclosure for the Fire Sensor Project we wanted to ensure that it would withstand fires and also ensure the model would look like a smoke detector alarm. The model was to ensure it would blend into buildings the way smoke detectors do. Through Solidworks we designed and 3-D printed an enclosure that perfectly fit all the components. In this enclosure there were exact spots for the radar, RealSense camera, and Jetson nano. The enclosure accounted for plugging in the devices to the wall and protected all of the devices. The enclosure demonstrated to have a sleek look and functioned well.

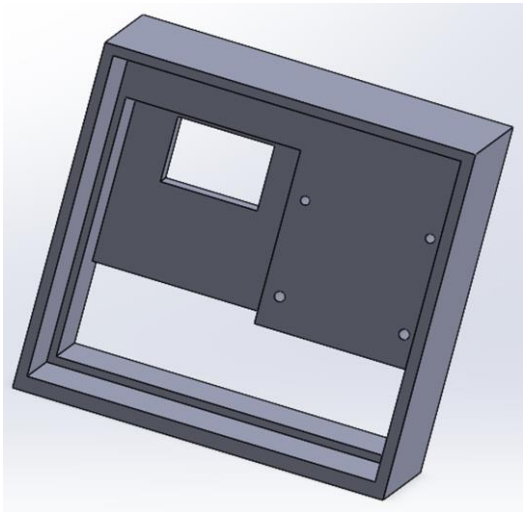

Fig. 11 3D model of enclosure

Fig. 12 Final Product

TESTING SET UP

Before we began testing, we set up our testing apparatus. The apparatus consisted of a 5-sided acrylic box, a piece of cardboard to cover the top, a pipe, and a fog machine. We put our prototype inside of the acrylic box which was filled with fog. The fog was pumped through the pipe and the apparatus was closed by a cardboard box to prevent any of the fog from escaping. We used this to emulate a room filling up with smoke without filling up the entire room.

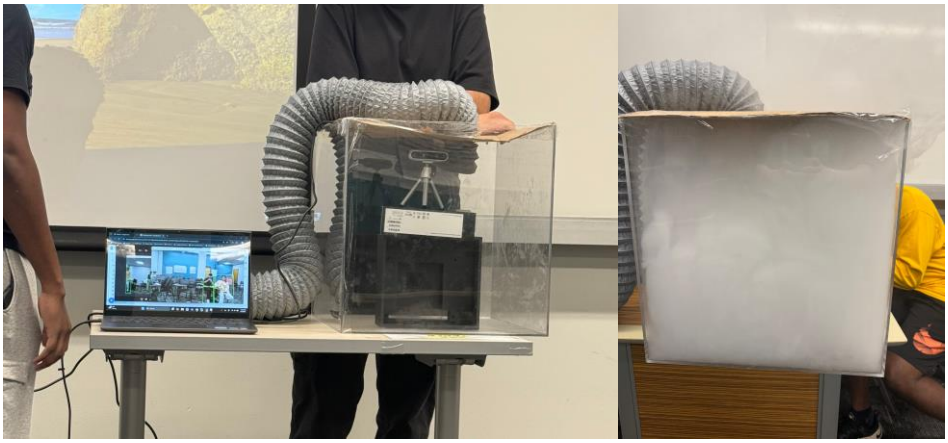

Fig 13. Testing Apparatus with Component Inside

The first step was to find a human detection model robust enough to detect individuals across varying lighting conditions, poses, and environments. The selected pre-trained model was trained on a dataset containing 8,091 annotated images, achieving a precision of 90.6%, recall of 87.9%, and a mean Average Precision (mAP) of 93.7%. In this context, *recall* refers to the proportion of actual positives correctly identified by the model, *precision* indicates the proportion of true positives among all detected positives, and *mAP* represents the mean of average precision scores across all detected classes.

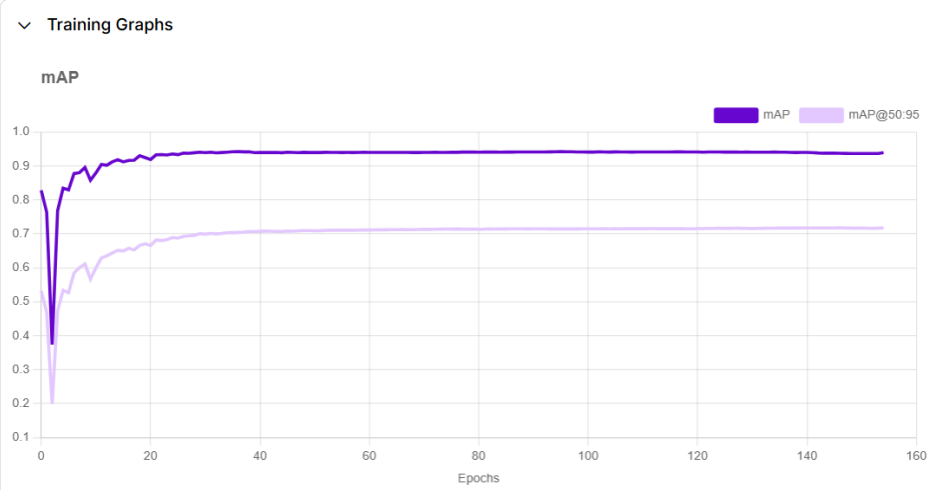

Fig. 14. Graph of mAP results across training epochs

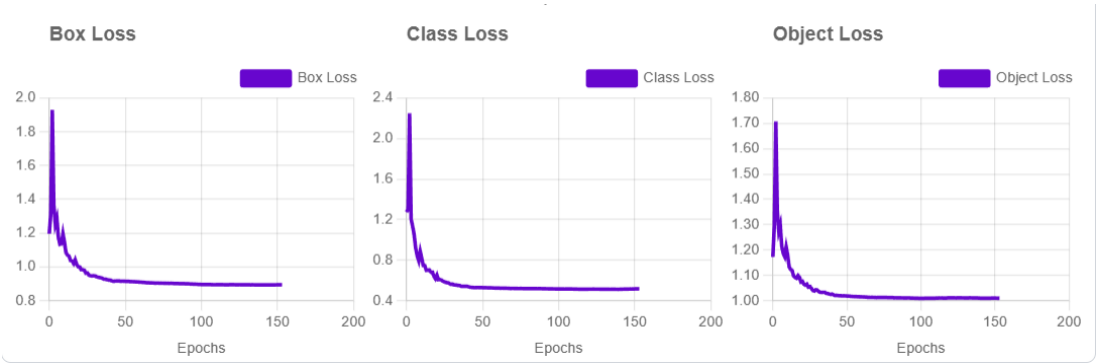

Fig. 15. Box loss, class loss, and object loss during training

To improve the model's performance under adverse conditions, an additional 771 images were captured and manually annotated. These images featured people under low visibility settings, with varying brightness and exposure levels, simulating environments like smoke-filled areas. These enhancements aimed to increase the model's robustness in scenarios with reduced image clarity due to smoke. Preliminary testing was conducted using the laptop's built-in webcam to validate the model's ability to detect individuals in challenging lighting and positional contexts. The model successfully identified multiple people, including partially hidden individuals.

Subsequently, the model was integrated and tested with an Intel RealSense camera. This involved configuring the RealSense pipeline using the Roboflow API, ensuring a stable 60 FPS video stream, and executing object detection inference in a parallel thread to maintain real-time performance. Once deployed, the system was evaluated under different conditions, as illustrated in the images below:

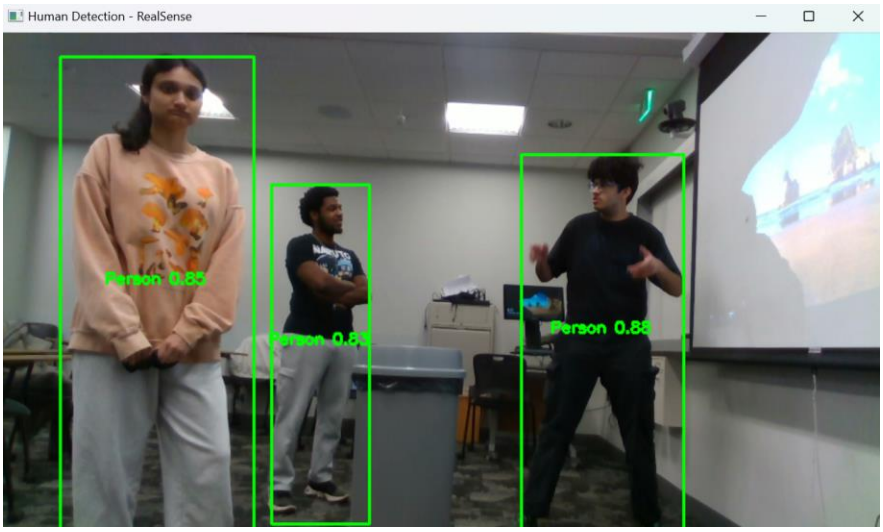

Fig. 16. Detection of multiple individuals in a single frame

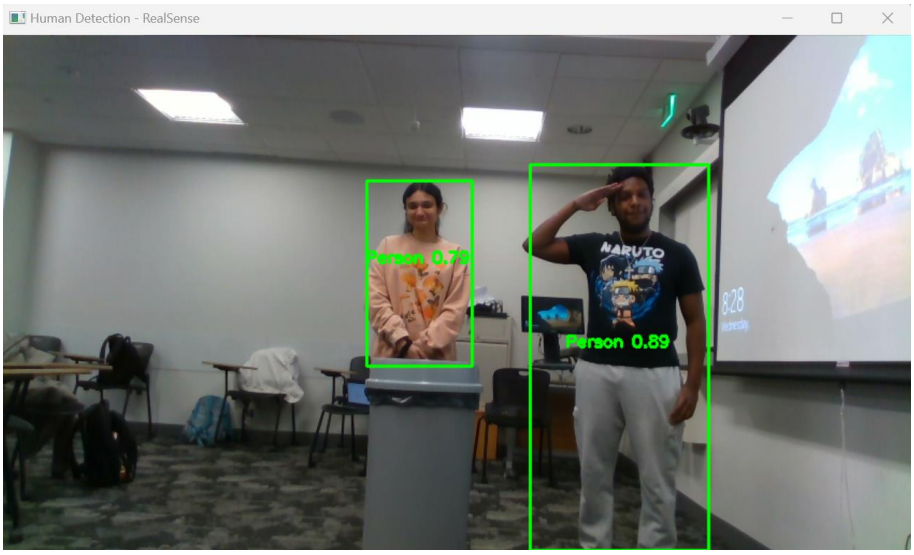

Fig. 17. Detection of one fully visible and one partially obscured individual

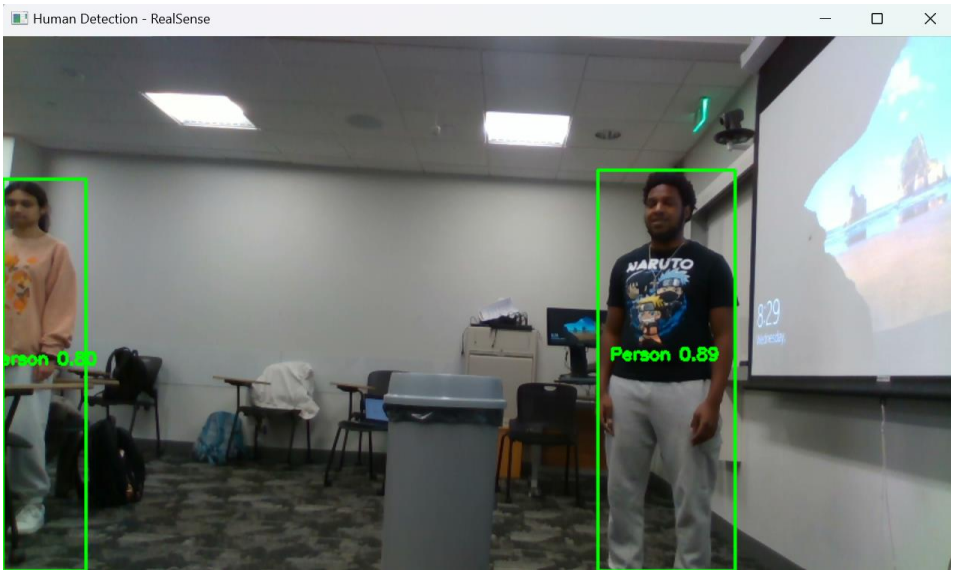

Fig. 18. Detection of a individual partially outside the frame

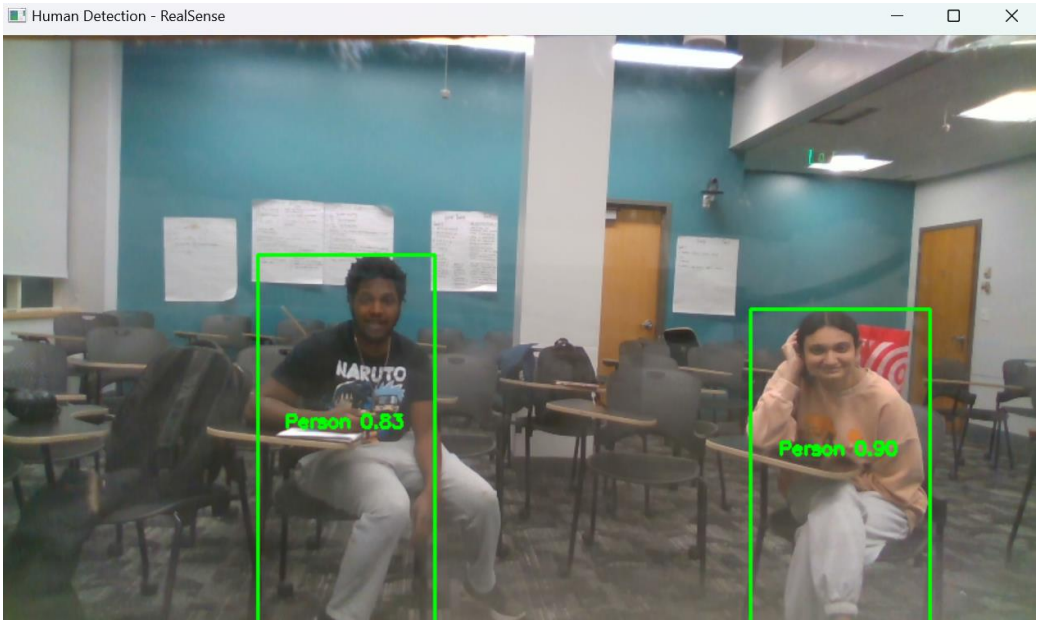

Fig. 19. Detection of a multiple individuals in minimal smoke conditions

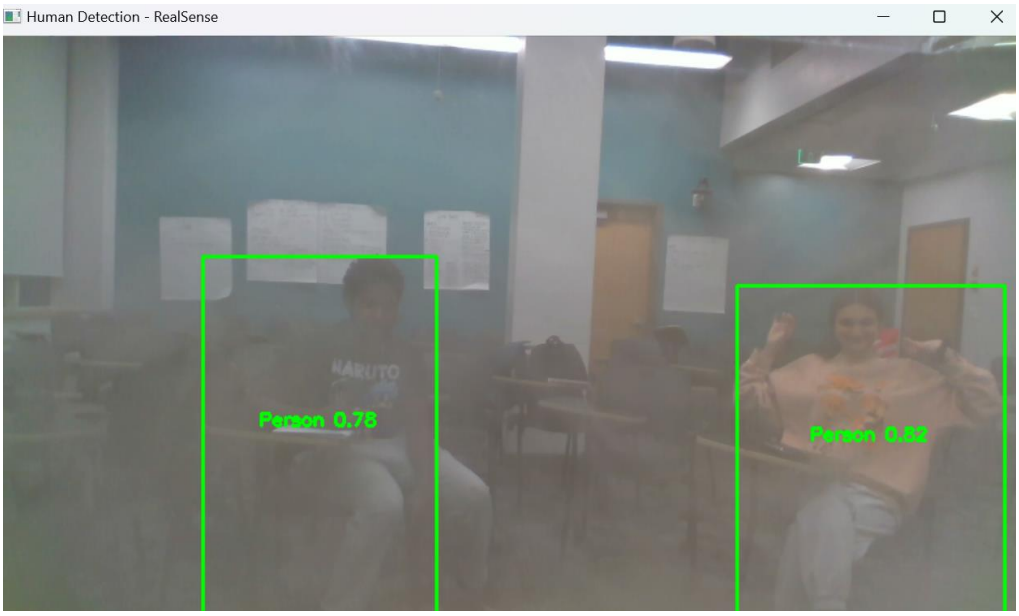

Fig. 20. Detection of a multiple individuals under heavy smoke conditions

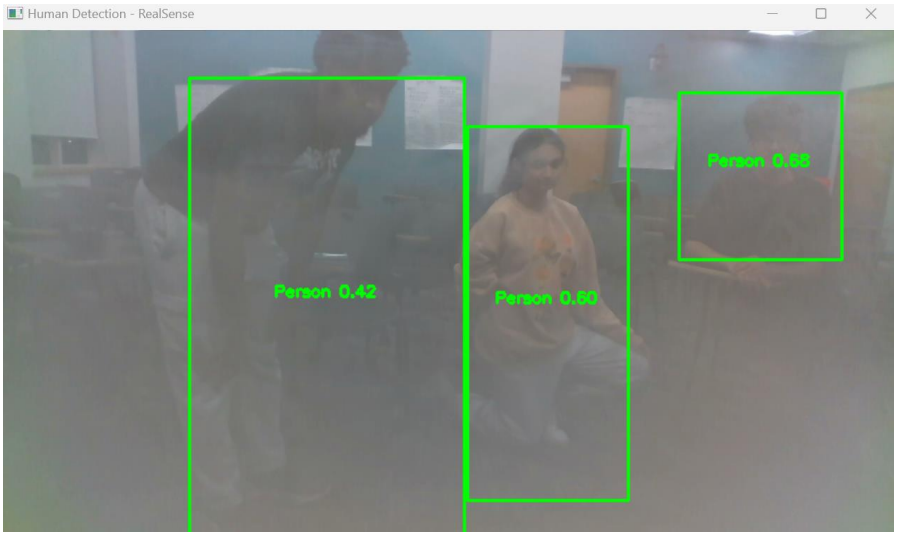

Fig. 21. Detection of a multiple individuals in varying positions under heavy smoke conditions

The green bounding boxes are generated through inference, where the model analyzes each video frame to locate people based on its training. Each box includes a confidence score, typically between 80–99% in clear conditions. Even when individuals are partially out of frame or obscured, the model maintains confidence levels around 65-80% still successfully detecting human presence. When we introduced smoke into the environment, the model's confidence

scores dropped to 49-85%, since the smoke obscures key visual features, yet the model still detected people, demonstrating strong generalization in low-visibility conditions. Additionally, the model accurately identified people in various poses, demonstrating its ability to handle real-world, unpredictable scenarios.

Radar testing focused on two aspects, human recognition, and speed of mapping. For human recognition we started by doing base scans standing and sitting outside of smoke. The random forest algorithm would interpret these scans and give a reading on whether or not a person was present. A variety of scans were tested to match the scans that made up the algorithm including both standing, sitting, under 20 iterations, and above 20 iterations. Through this process we continued to add good scans to the algorithms library in hopes of improving accuracy. And confidence score. Our final algorithm was correct 8-10 times with a confidence score of 78%.

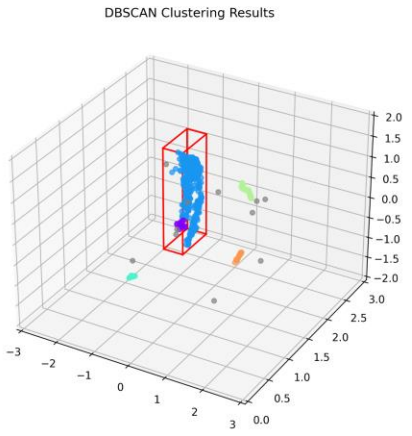

Fig.22. Non smoke DBScan

Fig.23. Human detected in DB Scan

We then took a similar variety of scans while spraying the radar with smoke. Some trials were attempted in the same testing apparatus as the real sense. However it was discovered that the acrylic box would be detected blocking the human subjects. To circumvent this obstacle we removed the box. To ensure that the radar was still covered in smoke the entire radar would continue to be sprayed with smoke for the entire testing process. During smoke testing as expected the radar scans remained unchanged. This gives us the same results as non smoke testing of 8-10 with 78% confidence.

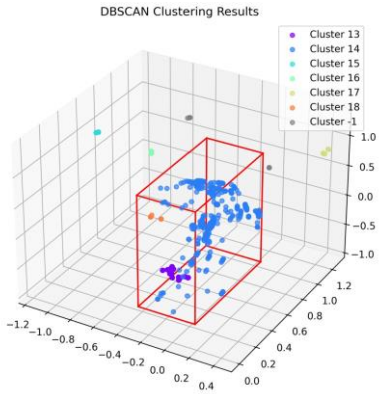

Fig.24. In Smoke DBScan

To test the speed of the radar we ran many trials on the entire product. We would then time how long it took for a room response and DBScan to be displayed on the website. Moreover we also tracked how long it took both of these to update. On average the time till image or image update would be from 8-14 seconds. Room status updates ranged from 6-12. The radar continues to run until the website buttons interrupt the function.

DEMO

For the demonstration, we acquired a closed room to represent the type of setting our prototype will be implemented in. Within this controlled setting, we installed the Intel RealSense camera, radar module, and Jetson Nano into our custom-designed and 3D-printed enclosure. We also set up the user interface to demonstrate the real time camera feed and point clouds of both the Realsense and radar.

Once the components were running, we demonstrated the prototype working in both clear and smoky conditions. To emulate the room filling up with smoke, we used both the testing apparatus and free smoke. Both versions allowed us to demonstrate the way the whole project worked together. The user interface took the data from the radar and the Realsense to give a readable output. The demo allowed for the confirmation that the fire sensor project functioned efficiently and at real time.

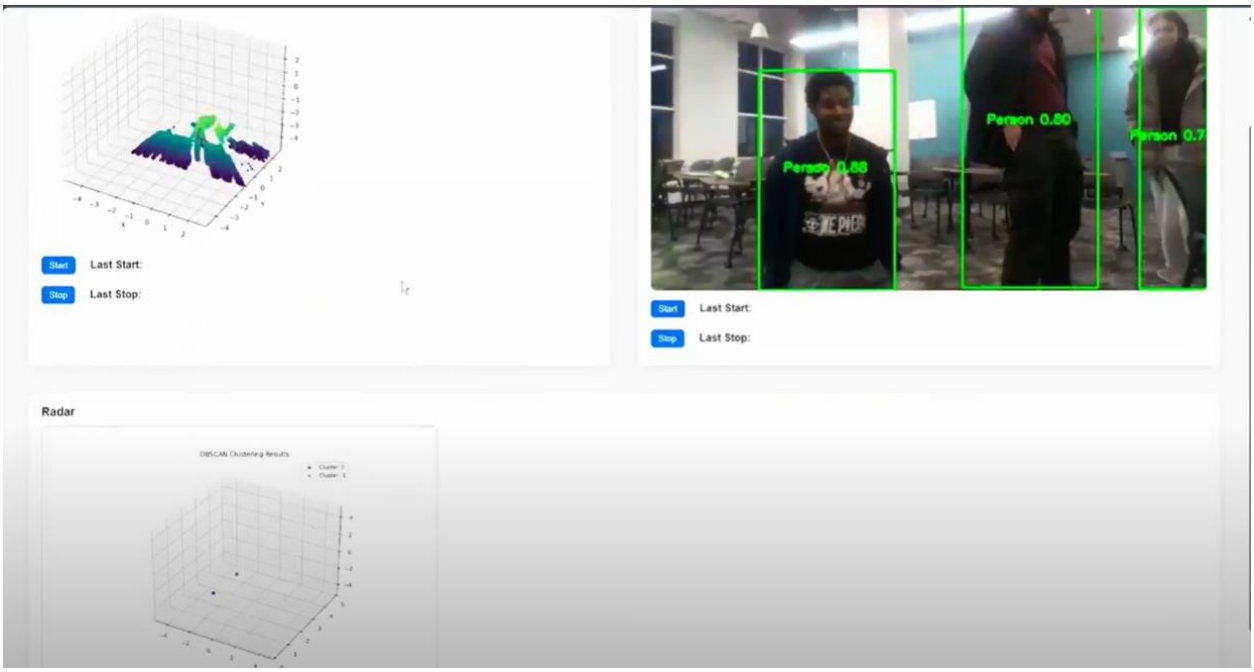

Fig. 25. User interface demonstration

COST ANALYSIS

The final cost was found to be far lower than the budget granted to us. The greatest expense was in the acrylic enclosure used to test the final product. At $102.99, it is over rice as expensive as the second greatest cost of the fog machine. This equipment was necessary to create a controlled, nonreactive environment. Ultimately, all costs were generated by the testing phase of product development due to the need to prove device efficacy in high stress environments. Only the 3D printing material at $23.99 was physically included in the demo. What allowed for costs to be below $200 was Professor Martinez-Lorenzo's support. The highly technical core components of the fire sensor, including an artificial-intelligence powered computer, radar, and camera, were granted by the SICA lab. Below is a chart itemizing the costs of the sensor's development.

Resource Description	Estimated Cost
mmWave Cascading RADAR	$0.00
RealSense Camera	$0.00
Jetson Orin Nano	$0.00
Smoke Machine	$39.99

Smoke Machine Liquid	$19.99
HVAC insulated pipe	$8.49
3D printing polymer	$23.99
Acrylic testing box	$102.99
TOTAL	**$195.45**

Considering the costs incurred by the department before the components were borrowed, the cost climbs to $2534.01. Such a price tag would be impractical in a commercial setting. It should be noted that the choice to use the mmWave radar, Jetson Nano, and RealSense camera were rooted in convenience and accessibility. Our device's low maintenance design means that far cheaper models can be implemented as the below devices were purchased for far more complex applications.

Resource Description	Estimated Cost
mmWave Cascading RADAR	$1318.80
RealSense Camera	$308.77
Jetson Orin Nano	$710.99
Smoke Machine	$39.99
Smoke Machine Liquid	$19.99
HVAC insulated pipe	$8.49
3D printing polymer	$23.99
Acrylic testing box	$102.99
TOTAL	**$2534.01**

CONCLUSION

The past four months saw incredible achievements in technical design, creative collaboration, and understanding of engineering ethics. After determining a desire to help a client base in need, this product emerged to help communities in need, especially without products close to the mainstream. Each segment of the team sought to insert a hardware, software, and ultimately humanitarian solution to a human focused problem. The integration of the SICA lab was vital to

our goals and allowed for greater integration of image processing and point cloud technology. The final project is an example of engineering that is human focused.

REFERENCES

[1] G. Jocher, “Yolov3,” Ultralytics YOLO Docs, https://docs.ultralytics.com/models/yolov3/ (accessed Apr. 17, 2025).

[2] N. Herrig, “Security camera monitoring with Computer Vision,” Roboflow Blog, https://blog.roboflow.com/security-camera-monitoring/ (accessed Apr. 17, 2025).

[3] U.S. Fire Administration, “Statistics,” *U.S. Fire Administration*, Sep. 27, 2022. https://www.usfa.fema.gov/statistics/

[4] “Roboflow,” About, https://roboflow.com/about#:~:text=Our%20Company,might%20prevent%20them%20from%20 succeeding. (accessed Apr. 17, 2025).