Senior Design Final Report

Mathworks

(Sensor Fusion for Autonomous Systems)

Team Members:

Hagop Arabian

Daniel Gallegos

Roberto Garcia

Gerardo Ibarra

David Neilsen

Patrick Emmanuel Sangalang

Jonathan Santos

Deepanker Seth

Angel Tinajero

Xiao Hang Wang

Faculty Advisor:

Dr. Manveen Kaur

Liaison:

Sumit Tandon

Table of Contents

1. Introduction

1.1. Background:

- In the world of technology, there exists a lot of companies who occupy that field. One of the companies is known as Mathworks. Mathworks is a company that utilizes mathematics and simulation through the use of their applications, Matlab and Simulink. Besides those two applications, they are also looking forward to sensor fusion for object detection and for lane changing. According to the National Highway Traffic Safety Administration, 10% of car crashes happen due to improper lane changes, resulting in 35,000 injuries, and 6,000 deaths per year in the United States Alone. This is very unfortunate to many, especially to those who have loved ones who will now grow up without those who have died. It may look like this situation is inevitable, but there is a solution that could possibly help us with the improper lane changes. We have created a sophisticated algorithm and a complimentary web application to show how it works, so it could take in data and use output from the algorithm to inform the user to make lane change.
- For the members of this project, we all contributed to different areas of the project. Some of the members contribute by programming a web application that will complement the algorithm for lane changing safety. Within that area, the work is divided into two: the backend and the frontend. One individual focused on creating the algorithm for the purpose of lane safety. Lastly, other individuals focused on the documentation to be able to write down information about the project while the other individuals work on their parts.
- How our project stood out from previous solutions was that while we focused on a simulation that involved using unity to see the animation, we decided to use each frame taken from the camera and put them in a way where it looks like a video. With each frame, the algorithm checks if it's safe to change lanes or not.

1.2. Design Principles

- With this project, the main framework is the algorithm itself. It will then be complimented by the web application that'll use the algorithm and show its capability. The web application will also be used to allow the user to see the frames as well as compare the video footage to the algorithm that detects the obstacles that are in the way. When the LIDAR and cameras see an obstacle that makes lane changing inefficient, the application will use either haptic feedback or vibrating to give a sign that you cannot make a lane change until that obstacle is gone from the side.

1.3 Design Benefits:

- By having our design like this, this would provide great benefits. By having the application give the ability for vibration or haptic feedback, we would save the lives of many and make lane changing much safer than before. It would also benefit with another advanced system called auto braking. Auto braking is a system in which the automobile would utilize the brakes automatically if there's ever an obstacle in the way before a collision could happen.
- Moreover, with the application, it would make accessing the algorithm and seeing the frames of each picture from the camera and LIDAR much easier, especially for those who are not tech savvy.

1.4 Results

- After months of working on the project and making the project work, we managed to create a web application that shows us each frame the driver and a comparison between the original video and the algorithm that checks if it's safe to change lanes or not based on if there's an obstacle or not.

2. Related works and technologies:

Lane changing is critical while driving on roads, and a slight miscalculation or mistake can potentially lead to major accidents. As the technology advances, the vehicle makers and engineers have implemented a number of solutions to assist drivers in making lane changes more safer and efficient.

2.1. Existing Solutions:

Existing solutions to lane changing assistance in vehicles include Blind Spot Monitoring (BSM) systems, Lane Change Assist (LCA) systems, and Lane Keeping Assist (LKA) systems among other from a range of automakers and industry experts.

1. Blind Spot Monitoring (BSM) Systems

a. Overview: BSM systems are designed to alert drivers about the vehicles in their blind spots while changing lane, using sensors or cameras on the vehicle's exterior.

b. Innovators: Volvo was one of the first companies to introduce BSM systems, with their Blind Spot Information System (BLIS) in 2005 (Volvo Car Corporation, 2005). Since then, numerous automakers have integrated BSM systems into their vehicles, including Ford, General Motors, and Honda.

c. Benefits and Limitations: BSM system increase driver awareness and reduce the likelihood of accidents caused by blind spots. However, these systems can sometimes produce false alerts, may not detect smaller objects like bicycles, and can be affected by adverse weather conditions.

2. Lane Change Assist (LCA) Systems

a. Overview: LCA systems go a step further by actively assisting drivers during lane changes. These systems utilize radar, ultrasonic sensors, and cameras to monitor the surrounding environment and, in some cases, can take control of the vehicle to execute a safe lane change.

b. Innovators: BMW was one of the first companies to introduce an LCA system in 2007, called the Active Cruise Control with Stop & Go (BMW Group, 2007). Other automakers, such as Mercedes-Benz, Audi, and Tesla, have since developed their LCA systems.

c. Benefits and Limitations: LCA systems can help prevent accidents by reducing human error and decision-making time during lane changes. However, LCA systems may not function optimally in poor weather conditions, and over-reliance on them can lead to complacency in drivers.

3. Lane Keeping Assist (LKA) Systems

a. Overview: LKA systems are designed to keep vehicles within their lanes by detecting lane markings and making necessary steering adjustments. Some systems also provide warnings to drivers if they begin to drift out of their lane.

b. Innovators: Honda introduced the Lane Keeping Assist System (LKAS) in 2003 as part of their Honda Accord model (Honda Motor Co., 2003). Many other automakers, including Toyota, Nissan, and Hyundai, have since developed their LKA systems.

c. Benefits and Limitations: LKA systems improve road safety by preventing unintentional lane departures, which can result from driver fatigue, distraction, or inattention. However, LKA systems may not function well in situations with poor visibility, such as heavy rain or snow, and can be less effective when lane markings are faded or not as clear.

Conclusion

In conclusion, existing solutions to lane changing assistance in vehicles, such as Blind Spot Monitoring, Lane Change Assist, and Lane Keeping Assist systems, have significantly improved road safety by reducing the risk of accidents related to lane changes. Innovators like Volvo, BMW, and Honda have played a pivotal role in developing these advanced driver-assistance systems. While these technologies still have limitations, they continue to evolve and will remain integral in shaping the future of autonomous vehicles. As the automotive industry progresses, it is crucial to continue refining these systems to further enhance safety and efficiency on roads.

2.2. Chosen Solutions Design and Justification

We used Cascade classifier which is available in OpenCV, a relatively easy to work with car detection algorithm, it requires lower computation resources (meaning we are able to port our algorithm into other platforms with not extensive computational resources).

We are also using Canny Edge detection which is available in OpenCV which is fast and requires lower computation resources.

Our vector magnitude formula is easy to understand and implement in a fast way.

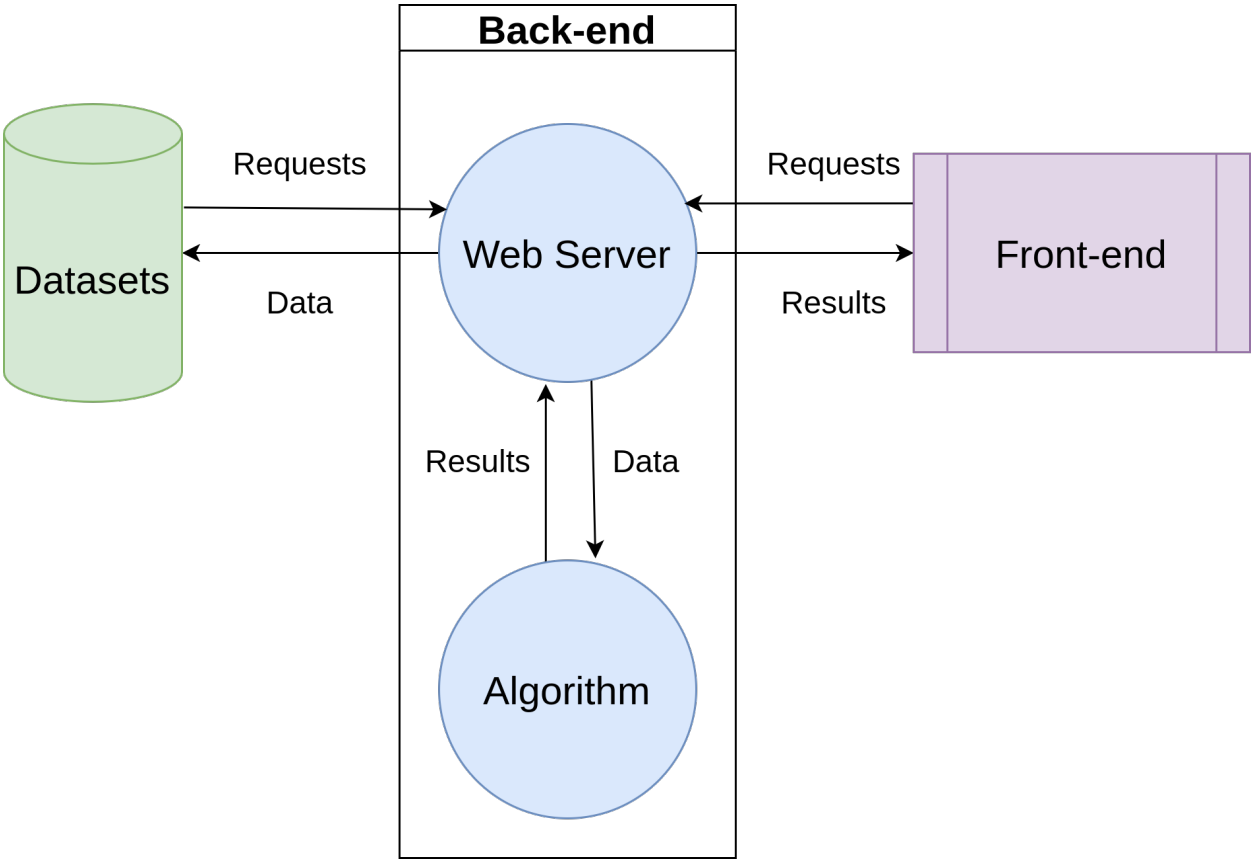
2.3.

- Cascade Classifier
    - The cascade classifier is one of the most popular and widely used object detection algorithms available in OpenCV. It has gained its popularity due to its high accuracy and low computational requirements. The cascade classifier is specifically designed for detecting objects in images and videos. It works by using a set of pre-trained classifiers to detect features of the object being searched for. These classifiers are arranged in a cascade, where each classifier is used to eliminate false positives that have been detected in the previous stage.
    - One of the major benefits of using the cascade classifier is its ease of use. It can be easily implemented and trained, making it an ideal choice for us who are new to the field of computer vision. Additionally, the cascade classifier requires lower computational resources than many other object detection algorithms, meaning that it can be easily ported to other platforms with less powerful computational resources.
- Canny Edge Detection
    - Canny edge detection is a popular algorithm for detecting edges in images. It is available in OpenCV and is widely used due to its speed and low computational requirements. The algorithm works by first smoothing the image to remove noise, then calculating the gradient of the image to identify areas with strong edges, and finally suppressing weak edges to obtain a thin line of strong edges.

    - The Canny edge detection algorithm is particularly useful in image processing applications that require real-time processing, such as robotics and autonomous vehicles. Its low computational requirements make it suitable for use on low-powered devices, such as embedded systems and mobile phones. Additionally, Canny edge detection is relatively easy to implement, making it an ideal choice for our project.
- Vector Magnitude Formula
    - The vector magnitude formula is a mathematical formula used to calculate the magnitude of a vector. It is commonly used in computer vision applications, such as image processing, motion detection, and object tracking. The formula calculates the length of a vector by taking the square root of the sum of the squares of its components.
    - One of the main advantages of the vector magnitude formula is its ease of implementation. It is a simple and straightforward formula that can be easily understood and implemented. Additionally, the formula is fast and efficient, making it suitable for use in real-time applications that require rapid processing of large amounts of data.

3. System Architecture

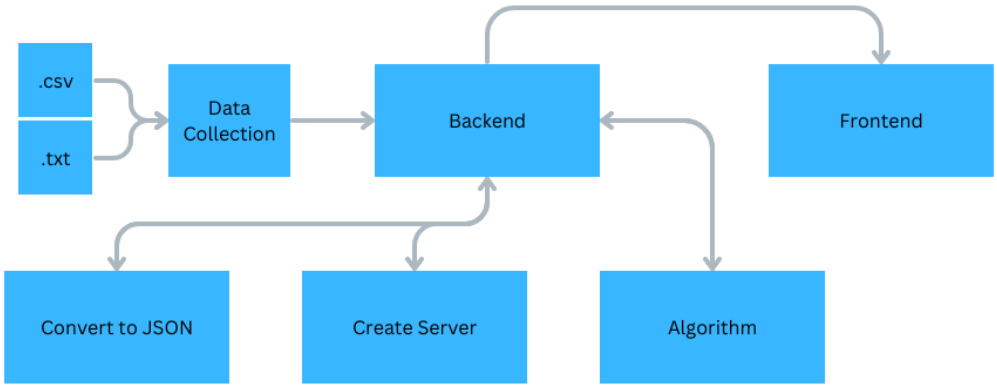
3.1 System Overview



The picture above is a general overview of the whole project. The project consists of a total of three major components, namely the front-end, back-end, and algorithms.

- The front-end is the part that interacts with the user and contains a drop-down menu that allows the user to select different pre-trained datasets and a graphical interface that displays the results of the algorithm.
- The back-end is a web server that receives commands from the front-end and invokes the algorithm to obtain the results, which are then passed to the front-end for display.
- The algorithm is the core of the project and consists of a series of functions implemented in Python that output a safe or unsafe result.
- The last remaining part is the dataset, which is currently from the KITTI Vision Benchmark Suite.

3.2 Workflow



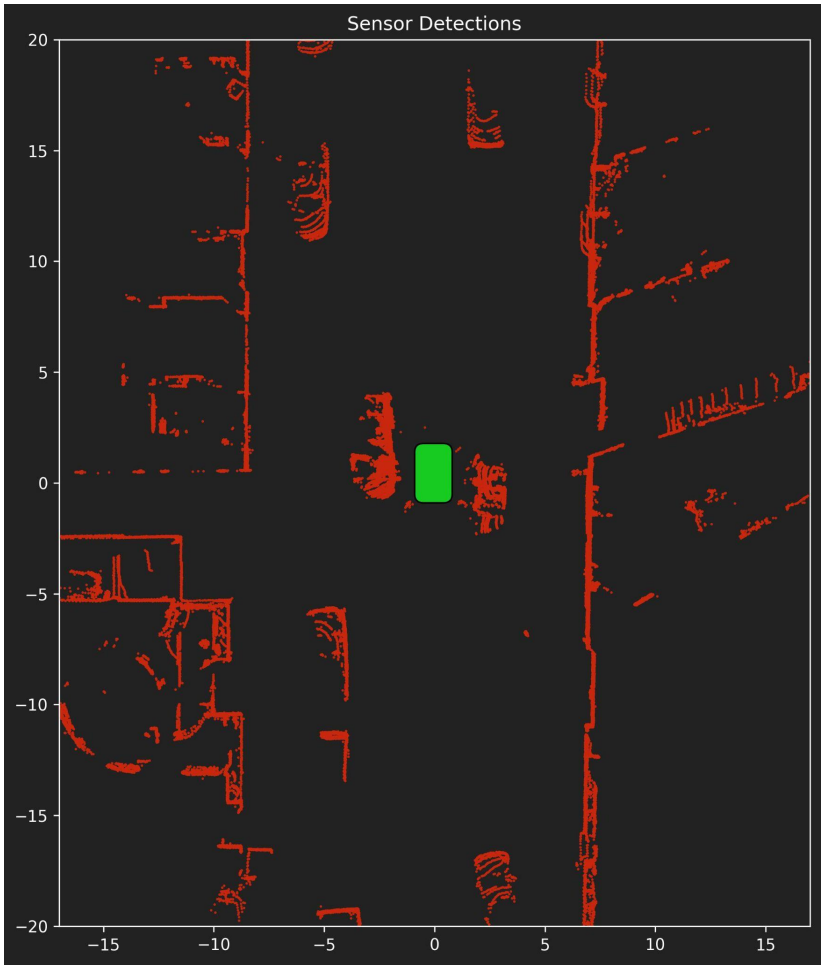
- In the original data, the lidar data is stored in csv and txt format, and the video frames are stored in png format.
- The backend reads the data and does the initial formatting, converting the data from the lidar into the data frame structure of the Pandas library and reading the video frames into memory via the OpenCV library.
- The backend takes the dataframe from the previous step and the images read by opencv as parameters to call the various algorithms in the algorithm.
- The different algorithms in the algorithm module store their results in JSON format.
- The Multi-layer Perceptron classifier in the algorithm module reads these JSON and passes them through the neural network layers to produce a final result, which is passed back to the back-end web server in the form of a function return value.
- The back-end web server replies to the front-end with the results from the algorithm in the form of JSON

3.3 Implementation:

- Front-end
  - Started with a React Image Comparison slider on a basic HTML webpage. Used to display the difference between a raw video frame and the same frame processed with canny edge detection.


```
<ReactCompareSlider
  itemOne={
    <ReactCompareSliderImage
      src={`http://127.0.0.1:5000/video_feed/edgless`}
      alt="Image one"
    />
  }
  itemTwo={
    <ReactCompareSliderImage
      src={`http://127.0.0.1:5000/video_feed/edges`}
      alt="Image two"
    />
  }
```

- A lidar plot was added next.



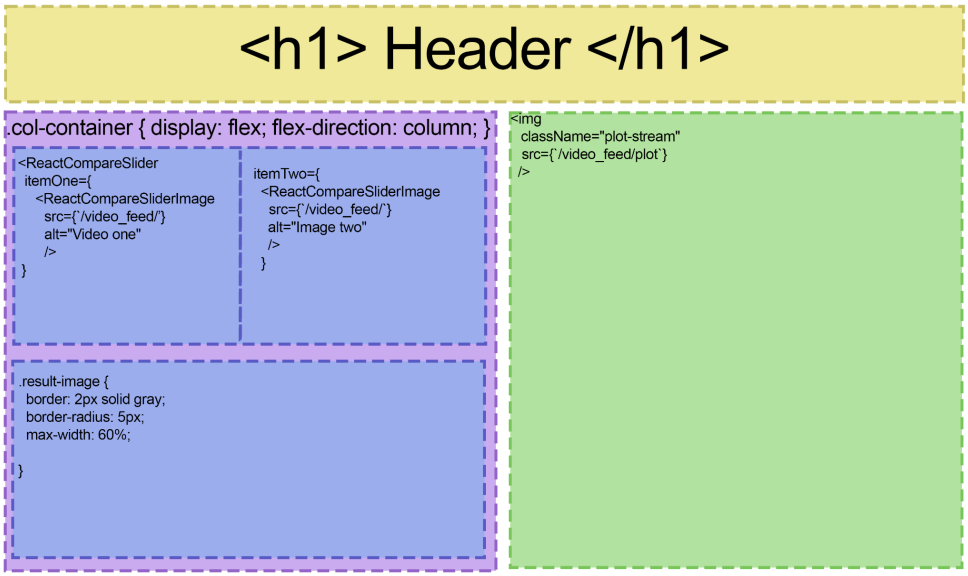
A Python program named generateLidarPlo.py was implemented to create the plots, leveraging the matplotlib library. The program would take the lidar detections data from the given dataset and mark a small red dot on the plot for each detection. The format of the plot was built with the following code snippet:

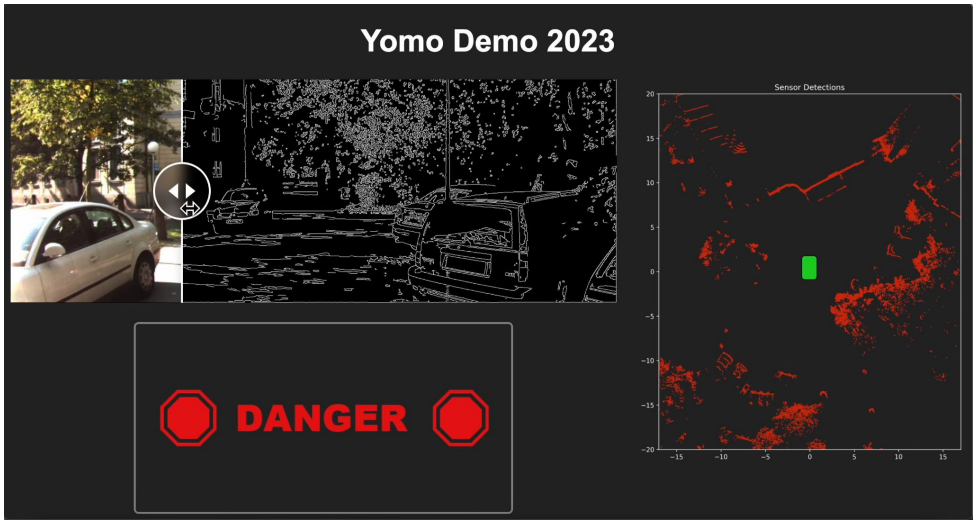
```
# Initialize the plot

fig, ax = plt.subplots(figsize=(10, 10))

ax.set_title('Sensor Detections')

ax.set_aspect('equal')



# Set dark theme background

fig.patch.set_facecolor('#222222')
```

```
    ax.set_facecolor('#222222')

    ax.spines['bottom'].set_color('white')

    ax.spines['top'].set_color('white')

    ax.spines['right'].set_color('white')

    ax.spines['left'].set_color('white')

    ax.title.set_color('white')

    ax.xaxis.label.set_color('white')

    ax.yaxis.label.set_color('white')

    ax.tick_params(colors='white')



    # Plot the car (a rectangle with rounded edges at the origin)

    car_color = '#17cc20'  # Set the color to the specified RGB
values

    car = FancyBboxPatch((-0.5, -0.5), 0.9, 1.88,
boxstyle="round,pad=0.4", fc=car_color, label='Car')

    ax.add_patch(car)



    # Plot the lidar points (top-down view)

    ax.scatter(-y, x, c='#c9270e', s=(0.3), label='Lidar points')



    # Shrink the plot range

    ax.set_xlim(-17, 17)

    ax.set_ylim(-20, 20)
```

- The last component was the warning output. Depending on the result of the algorithm at a given frame a ‘SAFE’ or ‘DANGER’ sign will be displayed.

- The final layout is organized with CSS flex boxes. The video comparison and result output being in one column and the lidar plot in another.





- Back-end
  - Started as a basic Flask webserver.

```
from flask import Flask, jsonify, send_from_directory, Response

import os

import json

from flask_cors import CORS
```

- The first routes implemented were primarily to serve jsons or individual frames from the dataset:

```
@app.route('/raw/<path:path>')

def send_report(path):

   return send_from_directory('raw', path)



@app.route('/')

def get_jsons():

   jsons_folder = './jsons'

   json_files = [f for f in os.listdir(jsons_folder) if
f.endswith('.json')]

   return jsonify(json_files)



@app.route('/frame/<file>/<time>')

def get_frame(file, time):

   jsons_folder = './jsons'

   json_file_path = os.path.join(jsons_folder, file)

   if not os.path.isfile(json_file_path):

       return jsonify({'error': 'No such JSON file.'}), 404

   with open(json_file_path) as f:

       try:

           data = json.load(f)

           value = data[time]

       except (KeyError, json.JSONDecodeError):

           return jsonify({'error': 'No such time in this JSON
file.'}), 404

   return jsonify(value)
```

In this form the web server could only serve single, still images.

- The next feature to be implemented was the /video_stream route which would serve simultaneous video streams to the front end.

```
@app.route('/video_feed/<path>')

def video_feed(path):

       return Response(gen_frames(path),    mimetype='multipart/x-
mixed-replace; boundary=frame')

def gen_frames(edge):

   frame_rate = 30  # Adjust this value to control the frame rate
of the video feed

   img_files = sorted(os.listdir(image_folder_path))

   num_files = len(img_files)

   i = 0

   while True:

       img_path = os.path.join(image_folder_path,
img_files[i%num_files])

       i += 1

       frame = cv2.imread(img_path)

       if frame is None:

           continue

       ret, buffer = cv2.imencode('.jpg', frame)

       frame = buffer.tobytes()

       yield (b'--frame\r\n'

              b'Content-Type: image/jpeg\r\n\r\n' + frame +
b'\r\n')

       time.sleep(1 / frame_rate)
```

Unfortunately this approach produced independent video feeds which did not sync up.

- In order to sync up the video streams a multithreaded architecture was implemented. In this approach:

    Each video_feeed() thread is given an ID.

    After a thread produces a frame it alerts all other threads.

    Goes to sleep until it's turn to produce another frame.

  This approach utilized the Lib/threading.py package for thread management, the time module for syncing, the itertools package to generate the thread ids and several global variables.

```
import time

import threading

from itertools import count



gen_frames_id_generator = count() # Used to
generate unique IDs for each thread

max_gen_frames = 4 # Raw Video, Edge Detection,
Lidar, and Results

num_gen_frames = 0 # Current number of threads

gen_frames_counter = 0 # Used to schedule the
order of the frames

gen_frames_condition = threading.Condition() #
Used to synchronize the threads
```

- The final implementation required a special stream to send the proper 'SAFE' or 'DANGER' signs based on the results from the algorithm:

```
@app.route('/video_feed/<path>')

def video_feed(path):

   global max_gen_frames

   global num_gen_frames

   # Check if we have too many video feeds

   if num_gen_frames >= max_gen_frames:
```

```
        return jsonify({'error': 'Too many video feeds.'}), 429

    else:

        gen_frames_id = next(gen_frames_id_generator)

        num_gen_frames += 1

        if path != 'results':

            return Response(gen_frames(path, gen_frames_id),
mimetype='multipart/x-mixed-replace; boundary=frame')

        else:

            return Response(gen_results(path, gen_frames_id),
mimetype='multipart/x-mixed-replace; boundary=frame')

# This function is used to generate frames for the video feed

def gen_frames( path, gen_frames_id ):

    global gen_frames_counter

    global num_gen_frames

    global image_folder

    image_folder_path = os.path.join(image_folder, path)

    frame_rate = 30  # Adjust this value to control the frame rate
of the video feed

    proper_wait_time = 1 / frame_rate

    img_files = sorted(os.listdir(image_folder_path))

    num_files = len(img_files)

    i = 0

    while True:

        start_time = time.time()

        img_path = os.path.join(image_folder_path,
img_files[i%num_files])

        i += 1

        frame = cv2.imread(img_path)

        if frame is None:

            continue
```

```
        with gen_frames_condition:

            while gen_frames_counter % num_gen_frames !=
gen_frames_id:

                gen_frames_condition.wait()  # Wait for our turn

            ret, buffer = cv2.imencode('.jpg', frame)

            frame = buffer.tobytes()

            yield (b'--frame\r\n'

                   b'Content-Type: image/jpeg\r\n\r\n' + frame +
b'\r\n')

            gen_frames_counter += 1

            gen_frames_condition.notify_all()  # Notify all waiting
gen_frames functions

        end_time = time.time()

        if end_time - start_time < proper_wait_time:

            time.sleep(proper_wait_time - (end_time - start_time))
```

- Algorithm
    - Longest implementation cycle of the three
    - Canney Edge Dection implementation
        - It invokes the Canny Edge Dection algorithm from the OpenCV library to save and return the edges.

```
CANNEY_EDGE_DECT(input_folder):

        result = []

        for files in input_folder:

                edges = OpenCV.Canny()

                os.save(edges)

                result.append(edges.pixels)

        return result
```

    - Cascade Classifier implementation

- It invokes the Cascade Classifier algorithm from the OpenCV library to detect vehicles and return the result.

```
CASCADE_CLASSIFIER(input_folder):
        result = []
        for files in input_folder:
                result.append(OpenCV.CascadeClassifier())
        return result
```

- Object Detection implementation
  - It calculate the magnitude of all the lidar data points and return if there is any object with a given range.

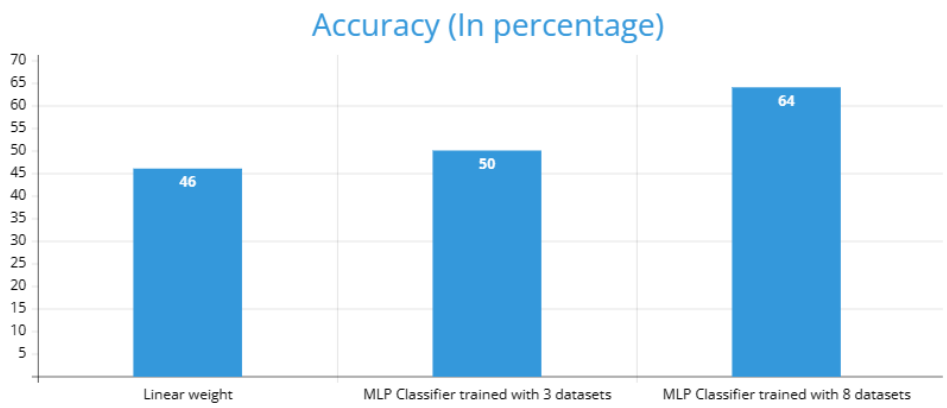
```
OBJECT_DETECTION(range, point_cloud_file):
        result = []
        for line in point_cloud_file:
                x, y, z = line[0], line[1], line[2]
                mag = Math.sqrt(x^2 + y^2 + z^2)
                If mag > range:
                        result.append(true)
                else:
                        result.append(false)
        return result
```

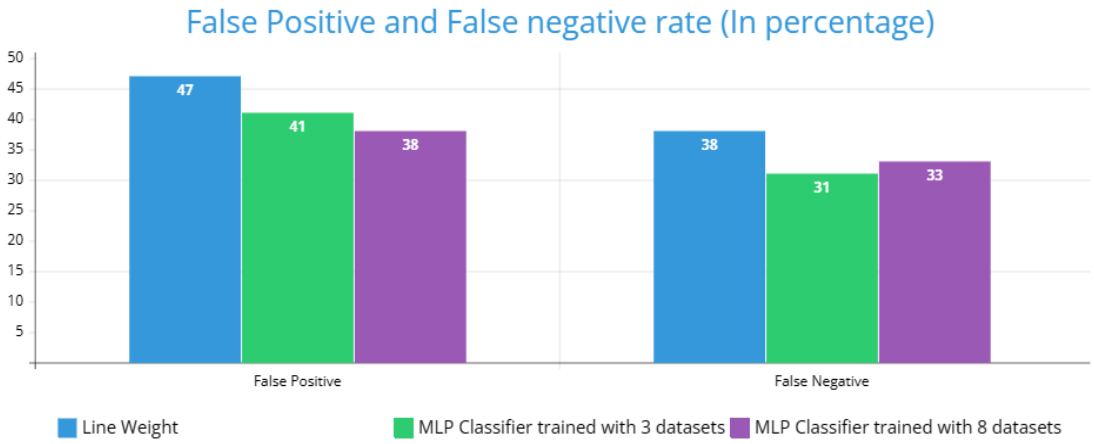
- MLP Classifier implementation
  - It loads the pretrain model (model.pkl) and fuses the results from the above three algorithms to get one final result.

```
MLP_CLASSIFIER(previous_results_file):

        model = Sklearn.MLPClassifier('model.pkl')

        X = previous_results_file

        final_result = model.predict(X)

        return final_result
```

4. Results and Conclusions

4.1. Results Overview





As the results show, our algorithm found the best possible accuracy when we used multiple datasets. The accuracy using the linear weight was the lowest of the three and the MLP Classifier training with 8 datasets had the best accuracy. This was a clear indication that the more datasets we use the better the accuracy would be. The false negative and false positive rates, shown in percentages, show that the training being done using the MLP Classifier with 8 data sets will achieve the lowest false positive rates and the second lowest false negative rates.

4.2. Successes and Failures

The area where we felt we found the most success throughout the project was the proficiency of our algorithm. The algorithm had decent percentages with the limited time we had to train it and gave us, relatively, good results.

The algorithm had an accuracy of:

- 46% - lowest using linear weight
- 64% - highest achieved using 8 datasets
- Approximately, achieved a 40% increase

The False Positives and False Negatives we found were:

- Highest FP and FN 47% and 38%
- Achieved 38% FP and 33% FN using 8 datasets
- Decreased rates by approximately 20%

4.3. Future Research and Development

The success of our project, and the algorithm's accuracy, can be improved immensely. Since we were testing and training the algorithm in different ways, we could not get the best possible accuracy. We found a curve trend when using the MLP Classifier using 8 datasets and believe that with an increase in training the algorithm can assist in the increase of the accuracy of the algorithm. An additional recommendation would be to train the algorithm for things such as: weather condition, road condition (e.g. potholes, etc…), and ensure all applications of the system follow traffic laws.

5. References

- The KTTI Dataset Geiger, Andreas, et al. "Vision meets robotics: The kitti dataset." The International Journal of Robotics Research 32.11 (2013): 1231-1237.
    - https://www.cvlibs.net/datasets/kitti/
- Aura, Proctor. “Technology Guide: What Is an Automatic Braking System?” *Proctoracura.Com*, www.proctoracura.com/automatic-braking-system-guide. Accessed 12 May 2023.
    - https://www.proctoracura.com/automatic-braking-system-guide
- National Highway Traffic Safety Administration (NHTSA): https://www.nhtsa.gov/