LiDAR-Based Environmental Object Classification System

Final Report

Team: sdmay24-31
Client: Ahmed Nazar
Advisor: Dr. Mohamed Y. Selim
Team Members:
Ella Rekow
Sachin Patel
Zachary Schmalz
Anuraag Pujari
Ryan Sand
Daniel Rosenhamer

Team Email: sdmay24-31@iastate.edu
Team Website: https://sdmay24-31.sd.ece.iastate.edu/

Revised: 05/01/2024

Table of Contents

List of Figures and Tables

Figures:

Tables:

Introduction

Problem Statement

There is a lack of 3D LiDAR object classification models that work with multiple LiDAR data formats, are scalable in terms of the data size, number of object classes, and the ease of adding features, and are efficient in real-time applications.

Intended Users and Uses

The outcomes of this project are prepared to cater to a diverse spectrum of industries, ranging from research specializing in visual classification to systems looking to gather data while allowing people's personal identifying information. It extends its impact to individuals exploring innovative applications of obstacle detection and avoidance, which has many applications in the field of robotics for path planning. Moreover, LiDAR object detection helps derive the number of objects, for example, the number of trees in a forest or students in a classroom, while needing to be more precise to grab identification features. The project's ripple effect extends to improving autonomous vehicles for identifying obstacles and avoiding collisions. There are also many drone applications for surveying land and identifying key objects for which this project is worthwhile. In addition to practical applications, there are plenty of benefits for research and academia that can result from the completion of this project. The area of LiDAR systems has needed more research into the application of this spreading technology. This project allows more individuals to explore applications in autonomous vehicles, drones, or other areas that could benefit from LiDAR classification systems.

Work in Context of Related Products and Literature

Despite the limited research, we found a few relevant papers for our project. One of these, Deep 3D Object Detection Networks Using LiDAR Data: A Review was written in 2019, explores machine vision and how it interprets the environment, which is crucial for decision-making. Similar to what we are trying to do, they also say that object detection, a cornerstone of machine vision, extends to 3D, offering detailed size and spatial data and enhancing detection systems' versatility. LiDAR, with its precise spatial capture and immunity to lighting conditions, emerges as a promising sensor for 3D detection. Their paper mainly focuses on the steps required in 3D object detection that are analyzed from a fundamental standpoint in this study, covering target definition, input LiDAR point cloud data, network structures, output encodings, and evaluation metrics. It also provides an extensive overview of current developments in deep 3D LiDAR object detection networks, encompassing 52 papers that discuss essential technologies.

Another paper we referenced was about BirdNet, which uses the KITTI Object Detection Benchmark because it is an innovative method for assessing performance on 2D detection, bird's eye view (BEV), and 3D detection tasks presented in the study. The two notable contributions are a novel concept for BEV cell encoding independent of distance and LiDAR device resolution and a 3D detection framework that uses BEV images as input to identify automobiles, bicycles, and pedestrians. Additionally, Livox has a repository on GitHub, Livox Detection-simu, intending to be a real-time identification model. It does, noticeably, have several issues and is no longer maintained. A final similar project is a 3d detection and tracking view using the KITTI dataset and another Waymo dataset. The paper's main contributions are a novel cell encoding proposal for BEV invariant to distance and differences in LiDAR device resolution and a 3D detection framework that identifies cars, cyclists, and pedestrians using a BEV image as input.

Revised Design

Requirements

Functional requirements:

- (1) Development of an object classification training model for LiDAR sensors
    - Object detection/classification accuracy of >75% within 500ms
- (2) Creation of a labeled dataset suitable for object detection
    - Ensure comprehensive labeling of multiple different object types
- (3) Compatibility assurance with various LiDAR sensor data types
    - Works with at least 3 data types, including: .lvx, .ply, .las

Resource requirements:

- (4) Sufficient computational resources for storing the collected data, training, and testing the model
    - 8 GTX 1080 Ti GPUs
        - Focus on ensuring high-quality data processing and model training within a reasonable training time
    - 16GB RAM required
        - Requires a minimum of 16GB RAM for efficient operation and reasonable refresh rates
        - Aim to facilitate smooth data processing and real-time model responses

UI requirements:

- (5) Published dataset is hosted on a service in an easily accessible/navigable manner
    - Host the dataset with an intuitive README or description
    - Ensure that users can easily retrieve the dataset

Performance requirements:

- (6) Classify objects with >75% accuracy
- (7) Identification within a 500 ms response time

Legal requirements:

- (8) Adherence to data privacy and ethical standards
    - Safeguard and maintain the confidentiality of personally identifiable information
- (9) Restrictions on collecting sensitive information such as personally identifiable information and compliance with IEEE standards

Maintainability requirements:

- (10) Necessitates ongoing model maintenance and documentation for future reference and collaboration
- (11) Scalability and adaptability to work with wider ranges of objects and larger datasets
- (12) Thorough documentation for ease of understanding

Testing requirements:

- (13) Rigorous testing and validation of the object classification system is also essential. Testing is covered more in the testing-specific section of the document

Engineering Standards

- ISO/IEC 20889:2018 - Preserve personally identifiable information
- IEEE 1588-2008 (PTPv2), PPS (Pulse Per Second)
- IEEE 1451 family of standards
    - These standards collectively create a framework for smart transducers and sensors.
- Loose Agile development process
    - Utilized Agile way of thinking for split teams and weekly progress assessments

Security Concerns and Countermeasures

Our LiDAR detection system will offer significant benefits in several areas, such as autonomous driving and 3D mapping, however, there are security concerns to be addressed for our project. Data privacy and ethical standards are at the forefront of issues that must be accounted for. LiDAR systems can collect vast data about the environment and the individuals within it. There must be some safeguards to maintain the confidentiality of individuals' personally identifiable information that may be captured during LiDAR operations. This means there should be restrictions on collecting sensitive information, such as facial details and license plates, to prevent misuse of said private data. Our project complied with standards set by the IEEE, so we can be sure that our LiDAR usage was responsible and ethical. These countermeasures can help mitigate the security and privacy risks that arise from our LiDAR detection, meaning our benefits can be accomplished without compromising security and privacy. Specifically, our data-collecting procedures did not include positions close enough to vehicles or people to record individually identifying features. We also collected data in public spaces outside the campus library and along sidewalks throughout campus.

Design Evolution

Our design has changed noticeably from the previous semester. The complete design can be found in the Implementation Details section. The main changes to our design begin in our first section which includes the data collection process of our design. We discovered that OpenPCDet does not require the syncing of camera data to the frames of the LiDAR, so we are now using our webcam data solely for helping to validate the LiDAR point clouds visually, especially during the labeling process using the MATLAB LiDAR Labeling tool. In the data processing section of our design, the details of how we pre-processed our data have changed. We are now using the Livox Robot Operating System Driver to convert between different data types, mainly .lvx to rosbag.

To achieve this, we created an Ubuntu 18.04.6 VM using Virtualbox to host the Livox ROS Driver repository and installation of ROS that matches it. We then utilized the conversion feature of the driver to transform our .lvx files to .bag files. This is done to allow for our following design change, which is using a semi-manual tool in MATLAB to label our data, to work. We ran into issues with the Point Pillar network and had to fall back on drawing frames manually at the start and end of objects and use an interpolation tool to draw the intermediate frames. A final change is being completely sure that we are using the OpenPCDet model as our project's base. Contributing to an existing model allows us to progress faster without creating an entire model ourselves. Also, it provides a way for more people to be able to use our contributions by attaching them to a preexisting and relatively well known model.

Implementation Details

Detailed Design

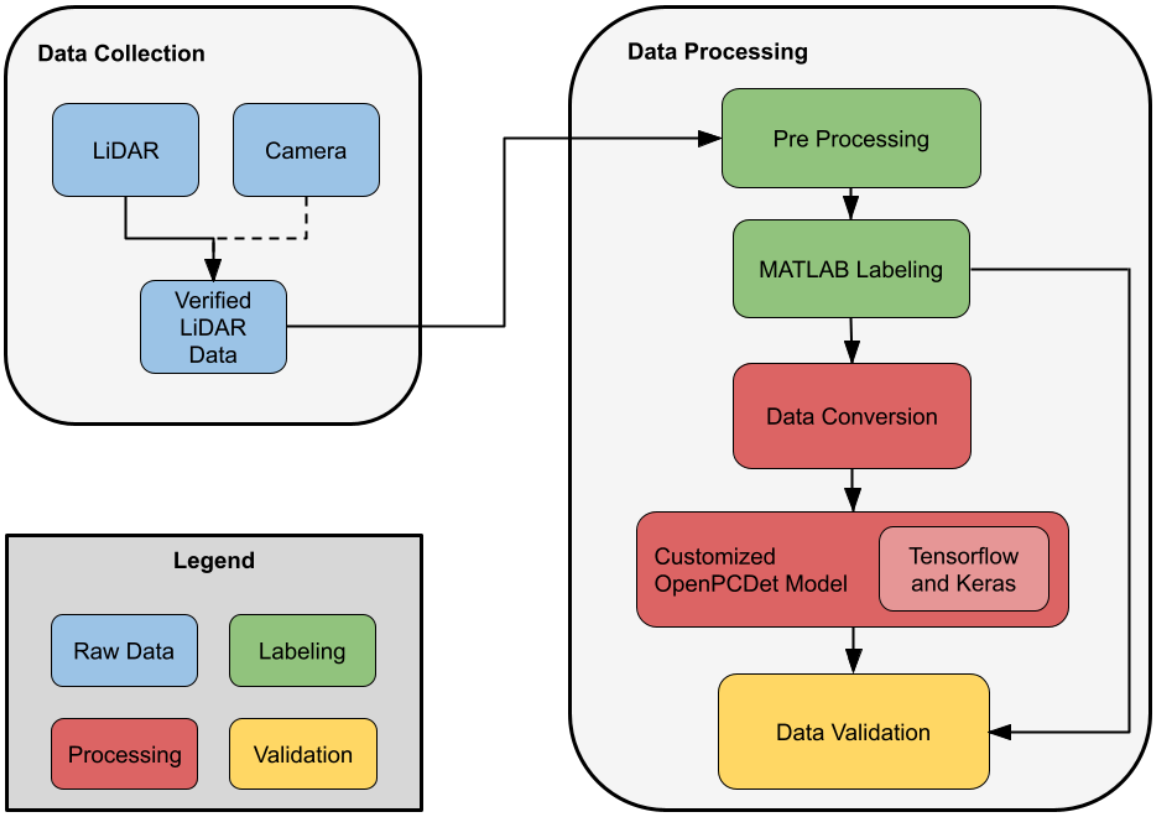

Figure 1: Final design iteration of the system.

Our design has gone through several iterations to reach its current point. The design showcases the process from start to finish of generating a successful result and testing it to ensure it is within our specifications. The final design, and almost every design iteration before this, has been split into two main sections. The above diagram shows the data flow through each of the varying steps of our project's process.

The first portion of our diagram is the Data Collection portion before reaching the first two components in this portion. In this step, we researched and brainstormed different locations for data collection across the Iowa State University campus. We considered the time of day and different angles possible to ensure that the data we collected would be helpful for both the testing and training of our learning model by finding a location with many objects to identify, such as cars and pedestrians. This helps the project meet requirements, particularly requirement 5 (External data will be collected around campus as needed), and must satisfy requirement 9 (Restrictions on gathering

sensitive information such as face details and license plates and compliance with IEEE standards). We collected the data using the Livox Mid-40 LiDAR provided to us by Iowa State University and a standard webcam. We also used the Livox Viewer tool on a laptop to connect to the LiDAR to stream the data to a .lvx file. We used a portable power supply to power the LiDAR and webcam and placed them on a tripod to sync their field of view as much as possible. Below is a series of synchronized frames in the Livox Viewer and the corresponding webcam data during data collection.

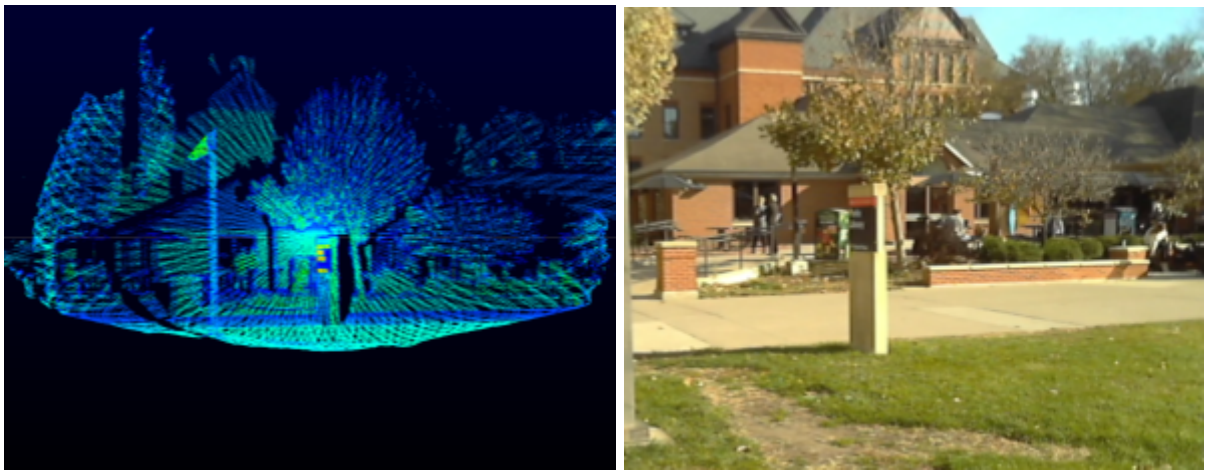

Figure 2: Livox Viewer combined frames of data on the left and corresponding webcam data on the right.

The first set of components in our block diagram visual data recorders. Once we decided where to collect data, we needed to physically collect it using a combination of LiDAR data to train and camera footage to help us label the LiDAR data more accurately. The amount of data we took at each location varied depending on the location and time. However, the length averaged around an hour. We then took that data to verify if the LiDAR data was clean and matched with the camera view. This is the first step in satisfying requirement 2 (Creating a labeled dataset suitable for object detection).

The labeling section is our second set of modules following the data recording components. We struggled in this section and eventually discovered a series of steps to achieve a labeled dataset. We first pre-processed our data by converting the point clouds into the correct data format, such as rosbag, using the Livox SDK tools. We then used an interpolation tool in MATLAB to manually label our data. This labeling technique was slower than we had hoped and did not allow automatic labeling. Instead of that technique working, a team member needed to draw a bounding box around an object when it appeared and when it disappeared from the frame to enable the tool to fill in the frames between them. Like the previous section of components, the labeling group of modules satisfies requirement 2 (Creation of a labeled dataset suitable for object detection). The

figure below showcases the user interface in the MATLAB LiDAR Labeling tool when placing a keyframe, either when an object enters the field of view or leaves. The left portion of the image shows the entire point cloud in one data frame. It is difficult to make out the scene due to how disparate the points can appear without movement. The yellow box is the proposed bounding box being manipulated to fit around a series of points that make up an object. The views on the right are different side or top views of the bounding box, allowing for easy resizing of the cuboid.

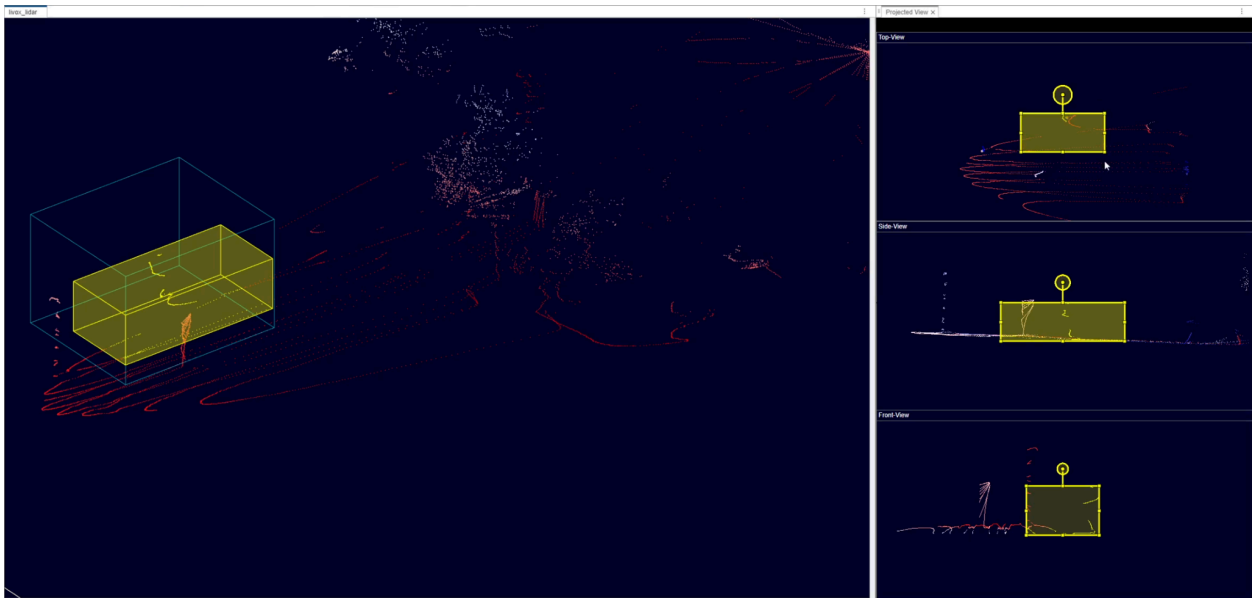

Figure 3: Manually labeling LiDAR data in MATLAB.

We also used the webcam data in this stage to help us determine where objects needed to be labeled, as the MATLAB representation of the point cloud was often difficult to parse. The following figure compares a frame in MATLAB to the same frame in the webcam data.

Figure 4: Viewing labeled pedestrians in MATLAB (left) and the webcam (right).

The next component group comprises our block diagram's processing and training portion. In our initial design, we planned to use tools like the You Only Look Once (YOLO) algorithm and the Tensorflow library to create the best model for our dataset. However, we moved away from using YOLO and instead have chosen to modify the OpenPCDet model. Before sending the data to the model for training, we needed to complete another step of data conversion. We wrote two Python scripts to translate the raw point cloud data in LAS form and the labeled data in a MATLAB exported file to the file formats OpenPCDet expected. Unfortunately, this is where we have ended with the project currently. We could not train the model with our data due to time constraints. This set of modules meets requirements 1 (Development of an object classification training model for LiDAR sensors), 2 (Creation of a labeled dataset suitable for object detection), and 3 (Compatibility assurance with various LiDAR sensor data types).

Our final step is the testing step, where the team will manually check the trained data against some data that we left explicitly for the checking step (20% for checking, 80% for training). This meets requirement 13 (Rigorous testing and validation of the object classification system are also essential) and will need to meet the milestone for the project to succeed. If we do not meet the milestone of 75%+ accuracy, the team will discuss and move back up the pipeline to a point further back and complete the steps again until our model is successful. This step was unable to be completed during the duration of our project due to challenges and time constraints, which will be discussed in the notes section.

Description of Functionality

We originally planned on creating a simple user interface allowing them to upload their data file and handle all the training and testing. This proved to be infeasible due to several factors. The first was that the labeling process was less automatic than we hoped. Beyond that, as we learned more about how models are implemented, we discovered that they are designed to be run in a more manual configuration. Finally, we ran into several complications in the model configuring process that prevented us from developing a tool of that scale during our project timeframe.

As a result of these reasons, our project operates using a series of separate programs and services as well as the modified OpenPCDet implementation. We wrote some of these, and others are tools created for specific purposes that we utilized in our process pipeline. Once Livox data has been collected, the Livox Viewer tool converts the .lvx file to a .las file. This file is transferred to our .las to .npy file converter script written in Python, creating a .npy file for each point cloud frame. These .npy files are used as data

inputs to the OpenPCDet model. The other input is made by using the Livox ROS Driver codebase to convert the .lvx file to a .bag file, which utilizes an installation of ROS. This .bag file is inserted into MATLAB and is labeled semi-manually using the LiDAR Labeler tool in the LiDAR Toolbox. The final output of this labeling process is fed through a custom converting script we wrote that takes in the exported MATLAB object file in a CSV format and creates the series of text files that contain the labels for each frame in which they exist. Finally, these files are added as the second input to the OpenPCDet model, which is then trained and tested on the data. The model process splits the data into 80 percent for training and 20 percent for testing.

This implementation is designed to meet all of our functional requirements. These include having the algorithm detect objects correctly with an accuracy greater than 75% within 500ms (1). It will also be scalable because of the OpenPCDet model, making it easy to add new objects to the system if necessary (2). Adding the data converting module will help ensure that our model can receive input in various LiDAR data types, including .lvx, .ply, and .las (3). Using the different tools we have at our disposal, including the Livox Viewer, the Livox ROS Driver, CloudCompare, and our scripts, we can convert between the most common file types for various LiDAR systems.

Notes on Implementation

Throughout the implementation of our design, we ran into several issues and made some changes as we discovered problems with our current implementation plan. Most notably, we could not automatically label our lidar data and struggled to find and maintain the necessary versions to run the model. We could not use the PointPillar network to label our LiDAR data automatically. Instead, we semi-manually labeled our data with MATLAB's Lidar Labeler and an interpolation tool, requiring us to create a Python script to parse and convert the LiDAR Labeler output into a format that OpenPCDet can use as input.

When beginning work with the model, we quickly found it necessary to utilize a remote desktop due to the required resources to run the model. This meant working with a clean Ubuntu install that each member could access remotely. From there, we needed to configure various dependencies, including CUDA, PyTorch, OpenCV, and more. Due to multiple factors, such as hardware limitations, interdependencies between tools, and the data we worked with, this was a very intensive task that involved a lot of trial and error. Due to this tool's limited documentation and small user group, running the model with our desired configurations took a lot of work. It played a large part in the challenges faced with the project's dependencies. This will be especially true when moving forward and working with the custom data set template provided in the project.

Testing

Process

With this project being the first year of work, most of our time was dedicated to building a solid foundation for this complex project. In addition, machine learning models aren't tested like conventional software. These circumstances prevented us from implementing many of the testing strategies outlined in this section. In our pursuit of deploying object classification models in machine learning, we recognize the need for a rigorous testing strategy inspired by the insightful paper by Eric Breck et al. at Google on ML production readiness and technical debt reduction. As we delve into the testing landscape for deep learning models, particularly those focused on object classification, we anticipate unique challenges stemming from neural network complexities and dynamic real-world data. Our future approach will integrate requirements, design considerations, and specific testing instruments to ensure production readiness and minimize technical debt. Drawing insights from the ML Test Score rubric, we aim to develop tests spanning data quality, model robustness, monitoring, and governance. By aligning our testing efforts with these critical facets, we strive to establish a resilient testing framework that guarantees our object classification deep learning model's reliability, adaptability, and interpretability in diverse production scenarios.

However, with future work focusing on testing the system as a whole, we have been able to verify our work even in its infant stages. Various tools have allowed us to determine if the environment we have configured is correct. This includes PyCharms dependency checkers, OpenPCDet's setup.py script, and error messages when running further operations. Additionally, using the KITTI dataset was crucial as the benchmarks posted on the front of the repository allow us to verify that our model works correctly.

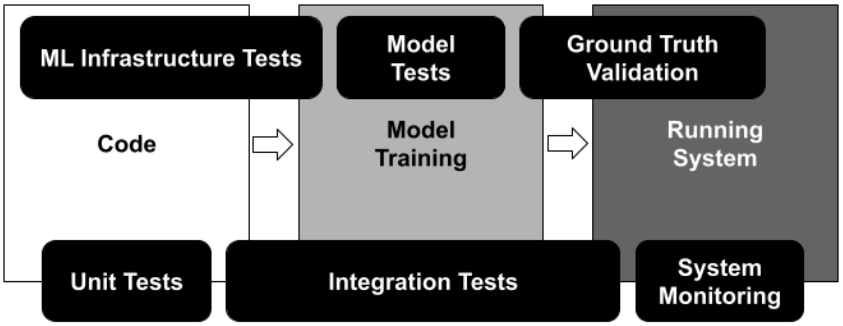

Figure 5: Our testing model

Figure 5: Machine Learning (ML) systems demand thorough testing and ongoing monitoring. A crucial distinction lies in the fact that, unlike manually coded systems (on the left), the behavior of ML-based systems is not easily predetermined. Instead, it hinges on dynamic aspects of the data and diverse choices made during model configuration.

Unit Testing

We understand that unit testing approaches provide particular difficulties when used with machine learning models because these models are data-driven and probabilistic. In contrast to deterministic programming, machine learning models depend on data patterns, which introduces uncertainty and makes their behavior reliant on particular datasets. We aim to develop a feature expectation schema that will allow us to compare data intuitions against input data in both the training and serving stages by encoding them into rules. We view data invariants as essential tools for performing dataset comparisons and schemas as early warning systems for any deviations. A deterministic unit test strategy will be aligned with a primary focus on reproducibility in model training. We are conscious, however, of the considerable difficulties in establishing perfect determinism in non-convex approaches. To guarantee reliability under various conditions, we will verify the model's specifications in the future—including input requirements, projected output, and performance indicators. By methodically incorporating model specifications into our testing framework, we hope to confirm that our machine-learning models fulfill the desired functions and function dependably in practical applications.

Interface Testing

Our future testing strategy will primarily use unit tests to validate the machine learning algorithm and guarantee its performance and dependability compared to predetermined criteria. However, we also recognize that interface testing is necessary to ensure that the neural network and data inputs operate together seamlessly. This testing will ensure the systems are adaptable and easy to use for automation or semi-automation by providing user-friendly data labeling and capturing procedures. While a complex machine learning algorithm user interface is outside our current project's purview, we will prioritize evaluating how well data is transferred across neural network stages. In the future, should time permit and the project scope grow, we plan to add more unit tests to assess the software that uses the algorithm, improving the overall robustness and usefulness of the system.

Integration Testing

We have identified two crucial areas in our future design plans that require our focus: ensuring our training data is quality-assured and efficiently training our machine learning model to recognize objects within LiDAR data. We acknowledge the critical importance of the latter, realizing that the caliber of the data our model is trained on impacts its performance. Therefore, we aim to perform comprehensive evaluations to reduce the possibility of overfitting or underfitting. We will primarily concentrate on the crucial route of model training, which entails moving data from the Livox Mid-40 LiDAR to the last phases of the machine-learning procedure. We want to use tools for sanity checks at each process stage and do human tests after the pipeline to guarantee data integrity and compliance with model requirements. To avoid data contamination and preserve data quality, we want to provide age limitations for data. We will watch and synchronize data streams using Matlab's manual testing to ensure the model doesn't overlook important information. Furthermore, automated checks will be implemented to confirm data accuracy and compliance with the necessary format. Additionally, we use a small sample of data to monitor and analyze anomalies closely, and we employ tools such as Tensorflow to perform thorough testing and debugging at every stage of the process.

System Testing

Going forward, we acknowledge that system testing is crucial in assessing a machine learning object categorization system to guarantee its dependability, resilience, and efficiency in practical situations. This thorough analysis goes beyond evaluating specific parts and concentrates on assessing the system as a whole to ascertain its capacity to manage a range of input fluctuations. The model's performance will be evaluated using critical metrics, including precision-recall and confusion matrices, which offer important insights into the model's correctness, accuracy, and capacity to classify examples accurately across various classes. Furthermore, we will prioritize hyperparameter optimization to improve the model's performance and stability and guarantee effective operation throughout time. Verifying that there are no resource leaks will increase the dependability of the model in real-world settings. In general, our future work will focus on extensive system testing to confirm that the machine learning object classification system is reliable and ready for use before deployment.

Regression Testing

Moving forward, we acknowledge regression testing as an essential quality control procedure in deep learning model building for object classification systems, emphasizing object detection in captured LiDAR data. It will be crucial to ensure that any adjustments we make to our model don't unintentionally cause new regressions or errors as it develops. As part of our testing strategy, we will systematically verify that the model retains its accuracy and dependability even after updates or improvements. Regression testing will be given priority, in particular, to ensure that the model continues to be capable of accurately identifying objects that it has previously detected. We will meticulously discover and fix any aberrations that suggest a decline in performance by contrasting the present projections with established benchmarks or ground truth data. Through this rigorous testing approach, stakeholders will feel more confident about the model's long-term dependability and adaptability for practical implementation, which will aid in creating the most effective and accurate object detection algorithm.

Acceptance Testing

Our following efforts will be focused on meeting our client's deep learning model criteria, which call for an object detection time of less than 500 ms and a minimum accuracy of 75% on our testing data. We will highlight our dedication to fulfilling these benchmarks by showcasing the model's object identification skills to the client during our planned, frequent demos during meetings. We make every effort to make sure that the performance of our unit tests satisfies our requirements, understanding their critical role in guaranteeing the project's success. Many functional needs, such as constructing a labeled dataset, developing the object classification training model, and deploying the object classification system, depend on reaching the 75% accuracy criterion. Given our access to several LiDAR sensors, we understand the importance of passing the Tensorflow integration test, which will verify interoperability with different LiDAR sensors. Furthermore, to evaluate prediction consistency in various scenarios, we will be slicing the data in the future. This will improve the overall fairness and dependability of our model's object identification.

Test Results

Inspired by the ML Test Score rubric and tailored to the complexities of deep learning models for object classification, we aim to improve our testing procedure and produce thorough outcomes that meet our project specifications. Unit testing will be the first step in our future approach. We will focus on our primary method of testing the data, using eighty percent of the data to train the model and twenty percent to test it based on OpenPCDet's results. Ensuring smooth data flow across neural network stages—even without an advanced user interface—will be the primary emphasis of interface testing. By addressing overfitting, underfitting, and data quality issues, integration testing will focus on crucial pathways such as the caliber of training data and the training procedure. Precision, recall, and confusion matrices will be used as primary metrics in system testing, offering a comprehensive assessment of model performance across various datasets. We'll methodically include every component to guarantee a thorough evaluation of generalization and adaptability. Model stability and effective resource management will be confirmed by verifying adjusted hyperparameters and minimizing resource leaks. Regression testing will be essential to validate the reliability of the model over time and ensure that changes preserve prediction quality. In conclusion, we believe that our following testing approach will successfully negotiate the intricacies of deep learning models for object categorization, exhibiting strong performance and satisfying requirements in various scenarios.

Broader Context

Area	Description	Examples
Public health, safety, and welfare	This project can assist safety by using LiDAR in several use cases, mainly vehicles and detection systems when detecting pedestrians or other vehicles on the road. The deep learning network could also be tweaked to identify different objects and help remove human workers from dangerous scenarios.	Increase road safety by detecting vehicles more accurately in systems such as stoplight car detection or even autonomous vehicles. Could identify if an accident occurred. Has the potential to monitor dangerous locations without human supervision, which would keep human operators safer while ensuring privacy.
Global, cultural, and social	Our project does not substantially impact these categories. However, some privacy concerns could be eased by implementing our project.	Replacing regular video cameras with a LiDAR system could improve the privacy of recorded individuals.
Environmental	There are no outstanding environmental impacts compared to the typical environmental damage caused by the manufacturing of LiDAR and related items. The laser is not dangerous because the LiDAR we use is a class I laser. However, there may be environmental applications for the object detection algorithm, potentially for positive impact.	Detecting the number of trees or animals in a forest over time. Observing erosion or other land alterations with concrete data.
Economic	This project will be available publicly and may save companies work hours for developing something similar. The cost of the system is comparable to a security camera system. It will likely provide consumers and companies with alternatives for regular cameras depending on their needs, as the software to detect objects is more readily available and more accessible to use. This is not a product we are selling, so while it might slightly improve LiDAR sales, it will not be marketed.	The learning model could entice businesses or researchers to use LiDAR systems, such as a regular camera setup instead of others. Because it will be made publicly available, the model could assist consumers with a lower budget because it will work with different LiDAR types

Conclusions

Review Progress

We have been able to make a significant amount of progress within our project. We began this year knowing that completing our project would be pretty ambitious. Despite this, we completed the tasks of collecting data, manually labeling objects within our recorded data, configuring the environment to be able to successfully run the OpenPCDet framework with a PV-RCNN model, creating a script to convert output LiDAR labels from MATLAB into a format acceptable to OpenPCDet, and training with OpenPCDet using Kitti data. This leaves the following contributors in an excellent position to continue adding to the highly beneficial framework that is OpenPCDet. With this, we are confident that we have contributed to allow us to let an extremely well tested and scalable model become more accessible.

Value of Design

Our project's primary concern is having an object classification system that can work with multiple LiDAR data formats and still perform when the amount of data increases. Our project can provide a tremendous amount of value towards accomplishing this goal. Contributing to OpenPCDet, which utilizes several models, allows individuals to choose a configuration that will best meet their needs with a varying size of data. The most significant piece of value that our project should provide is the ability for tons of people to continue to add more to the OpenPCDet framework, which can help its development into a more accessible and well-tested model. With the addition to the OpenPCDet framework, we hope other researchers will take this generic model from OpenPCDet and use our additions to work towards other more specific applications of LiDAR detection.

Potential Future Steps

Despite all the progress we were able to make this semester, unfortunately we weren't able to complete the project in entirety. Considering this, the following steps to our project will likely have to be completed by the next senior design group that picks up this project next year. More specifically, there will only be a couple of tasks that need to be completed before it can confidently be said that the project is in the final stages of verification and testing to guarantee its functionality is correct and working as intended regarding the requirements. These steps include the completion of running our personally collected data on the OpenPCDet model and confirming that the detection and classification aspects of the model work along with more than just the Kitti dataset; the first dataset was trained with the OpenPCDet model. If this can be accomplished, then the project is in great shape to be tested and applied to a more extensive variety of datasets, along with more changes to the configuration to increase efficiency. More datasets mean better guarantees of greater precision and accuracy when it comes to the classification and identification of objects from LiDAR detection.

References

[1] “Livox Mid-360 User Manual v1.2,” Jun. 2023.

[2] J. Beltran, C. Guindel, F. M. Moreno, D. Cruzado, F. Garcia, and A. De La Escalera, “BirdNet: A 3D Object Detection Framework from LiDAR Information,” in 2018 21st International Conference on Intelligent Transportation Systems (ITSC), 2018.

[3] Y. Wu, Y. Wang, S. Zhang, and H. Ogai, “Deep 3D object detection networks using LiDAR data: A review,” IEEE Sens. J., vol. 21, no. 2, pp. 1152–1171, 2021.

[4] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, “The ML test score: A rubric for ML production readiness and technical debt reduction,” in 2017 IEEE International Conference on Big Data (Big Data), 2017.

[5] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation,” arXiv [cs.CV], 2016.

[6] Hailanyi, “3D-Detection-Tracking-Viewer: 3D detection and tracking viewer (visualization) for Kitti & waymo dataset” https://github.com/hailanyi/3D-Detection-Tracking-Viewer

Appendix 1 - Operation Manual

LVX to LAS format:

1. **Requirements**
    a. Livox Viewer Installed
    b. Sufficient space to store .las file
2. **Open Livox Viewer**
    a. In the download directory, execute Livox Viewer.exe
3. **Begin the converting process**
    a. On the top left of the screen, next to File, click Tools
    b. In the drop-down menu, click File Converter
4. **Select the files to convert**
    a. Ensure the conversion is set to Lvx to Las
    b. Use the … button to navigate to the .lvx file you wish to convert
    c. Use the … button to navigate to the directory where you want to save the new file
    d. Name the file that you wish
5. **Finish the conversion**
    a. Click start
    b. The conversion will take place and save the file in the directory you specified with the name given

LVX to ROSbag format:

1. **Requirements**
    a. Ubuntu 16.04 System
    b. Livox_ros_driver repository cloned
    c. ROS installed that matches the requirements of Livox_ros_driver
2. **Initialization**
    a. Open the terminal in ../ws_livox and run $ catkin_make
    b. Then run $ source ./devel/setup.bash to set up the ROS environment
3. **Conversion**

a. Finally, run $ roslaunch livox_ros_driver lvx_to_rosbag.launch lvx_file_path:=""
b. Provide the path to the .lvx file you wish to convert in the quotation marks
c. The conversion will make the rosbag file in the same directory with the same name as the original file

MATLAB Lidar Labeler:

1. **Requirements**
    a. MATLAB 2023b or newer
    b. Lidar Toolbox (MATLAB Add-On)
    c. If LiDAR data is in ROSbag format
        i. MATLAB ROS Toolbox (MATLAB Add-On)
2. **Open Lidar Labeler**
    a. In the MATLAB *Command Window*, run the command *lidarLabeler*. This opens a new window called *Lidar Labeler*.
3. **Import LiDAR data**
    a. On the top-left, select *Import -> Add Point Cloud -> From File*, then choose the *Source Type* from the drop-down. Next, click *Browse* and allocate the lidar data file. Finally, select *OK*.
4. **Create new labels**
    a. On the left-hand side, there is a vertical pane titled *ROI Labels*. Under the title, click the *Label* icon, enter the label name and type, then click *OK*.
5. **Set up the labeling algorithm**
    a. In the *Algorithm* section located in the top-middle of the screen, click *Select Algorithm -> Point Cloud Temporal Interpolator*.
    b. Right below the *Select Algorithm* option, click *Configure Automation -> Start time to End Time*.
6. **Find objects of interest**
    a. On the very top-left, there are two tabs, *LABEL* and *LIDAR*. Select the *LIDAR* tab.

b. Using the play buttons on the bottom-middle of the screen, find the time an object enters the scene and adjust the *Start Time* to the *Current* time. Change the *End Time* to the *Current* time an object leaves the scene.
c. Once the *End and Start times* are set, return to the *LABEL* tab on the top-left and click *Automate.*

7. Label objects

a. While in the *Automate* mode, only label the specific object from which you found its start and end time.
b. Select the appropriate label you created on the left in the *ROI Labels.*
c. Move the time slider to the first frame and place the label on the object.
    i. With the label placed and selected in the *LIDAR* tab, click the *Projected View* button.
    ii. Using the Top-View, click and drag the circle above the box to change the label's heading.
    iii. Using the Top-View, Side-View, and Front-View windows, resize the label to encompass the object's point cloud.
d. Move the time slider to the last frame and place the label on the object.
    i. Repeat steps 7i through 7iii.
e. Optionally, move the slider to several frames between the first and last and label the object. This can improve the label's adherence to the object.
f. Finally, in the *AUTOMATE* tab, click *Run*. Play back the results to verify correctness.
    i. If the labels track the point clouds properly, click *Accept*.
    ii. Otherwise, click *Undo Run* and adjust the labels accordingly. Then click *Run* again.
g. After clicking *Accept*, find and set another object's start/end time and repeat the steps above.

8. Exporting labels

a. In the *LABEL* tab, click *Export -> To File* to save the labels.

b. To save your session, click *Save Session* (highly recommended) in the LABEL tab.

Converting MATLAB Labels to CSV Format:

1. **Requirements**
    a. MATLAB 2023b or newer
    b. Lidar Toolbox (MATLAB Add-On)
2. **Converting to CSV**
    a. In MATLAB, click *Open*, navigate to the labels exported from MATLAB labeler, and select the file.
    b. In the MATLAB *Command Window*, run the following

```
dataTable = timetable2table(gTruth.LabelData);
csvFileName = 'exported_data.csv';
writetable(dataTable, csvFileName);
```

    c. This will convert the labeled data into a CSV format readable by the Python script below.

Python Label Conversion Script:

3. **Requirements**
    a. Python 3.x or newer
4. **Install dependencies**
    a. The following dependencies are required to run the script: *csv*, *shutil*, *os*
    b. It is recommended to use Python's built-in package installer to install these
        i. From a command line, run *pip install csv, shutil, osv*
5. **Run the script**
    a. Place the *exported_data.csv* file generated from the previous instructions in the same folder as this script
    b. From a command line, run the script with the following command
        i. *python convertLabels.py*

c. The script will print the labels, create a folder titled *labels containing* the labeled data in text files in a format readable by OpenPCDet, and create a file named *labeled_data.csv* that includes the labels in an unnested CSV format.

OpenPCDet:

1. **Requirements:**
    a. **Dependencies**
        i. Linux (tested on Ubuntu 14.04/16.04/18.04/20.04/21.04)
        ii. Python 3.6+
        iii. PyTorch 1.1 or higher (tested on PyTorch 1.1, 1,3, 1,5~1.10)
        iv. CUDA 9.0 or higher (PyTorch 1.3+ needs CUDA 9.2+)
        v. spconv v1.0 (commit 8da6f96) or spconv v1.2 or spconv v2.x
        vi. Pcdet library and dependent libraries. It can be installed using the following command: `python setup.py develop`
    b. **Hardware**
        i. A CUDA-enabled GPU, i.e., a 1080 series GPU. Alternatively, the CPU can be used; however will result in reduced performance.
        vii. A large amount of storage is dedicated to this. We recommend at least 512GB of storage.
    c. **Data**
        viii. Point clouds formatted as NumPy arrays (.npy)
            1. We used the KITTI dataset, which had this already
            2. The data we collected was exported from .lvx to LAS, then to NumPy array using Python.
2. **Data Preparation**
    a. If using a custom dataset, convert point clouds from LAS to NumPy array format using the python .las to .npy converter script shown in Appendix 4.
    b. Upload the data set into the project and organize it into the file structure provided by OpenPCDet.

    File structure for a custom dataset:

```
OpenPCDet
├── data
│   ├── custom
```

```
|   |   |── ImageSets
|   |   |   |── train.txt
|   |   |   |── val.txt
|   |   |── points
|   |   |   |── 000000.npy
|   |   |   |── 999999.npy
|   |   |── labelsr
|   |   |   |── 000000.txt
|   |   |   |── 999999.txt
├── pcdet
├── tools
```

3. **Training**
    a. Specify the model you want to run, the number of epochs, and the batch size.
    b. Command to run training procedure:

```
python train.py --cfg_file ${CONFIG_FILE} --batch_size
${BATCH_SIZE} --epochs ${EPOCHS}
```

4. **Testing**
    a. Specify the model, batch size, and the number of epochs you want to evaluate.
    b. Also, specify what checkpoint of the trained model to evaluate. The option is to assess the latest or previously trained model version. Another option is to evaluate all checkpoints to make a performance curve.

`Test specific checkpoint:`

```
python test.py --cfg_file ${CONFIG_FILE} --batch_size ${BATCH_SIZE} --ckpt ${CKPT}
```

`Test all checkpoints:`

```
python test.py --cfg_file ${CONFIG_FILE} --batch_size ${BATCH_SIZE} --eval_all
```

Appendix 2 - Initial Versions of Design

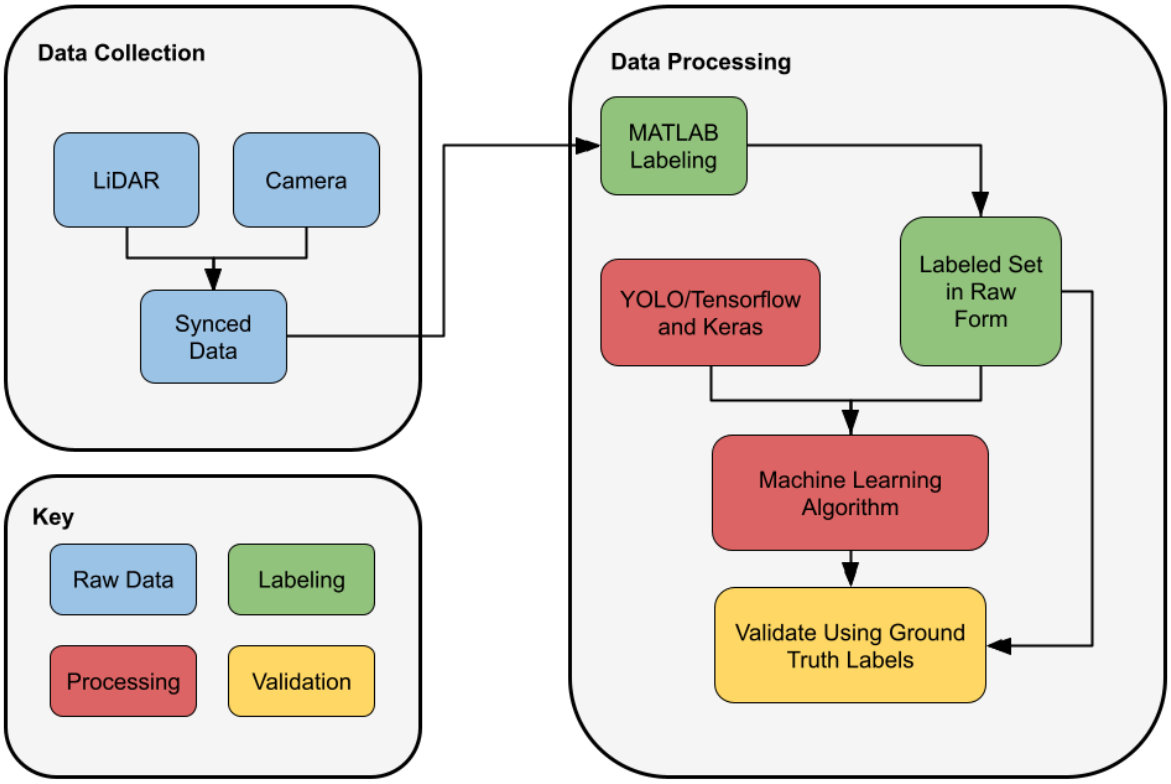

Figure 6: Initial design iteration of the system.

This diagram describes how we initially designed the project after the first few conversations with our client. We initially believed that we would create our machine-learning model from scratch. We eventually figured out that modifying an existing model and changing its configurations to work better with the Livox LiDAR data we were using would be much more feasible.

We were also not sure exactly what we would use for our machine-learning library and were considering YOLO. The group had been resigned to thoroughly labeling everything by hand in MATLAB, although we didn't fully understand what that meant at this time.

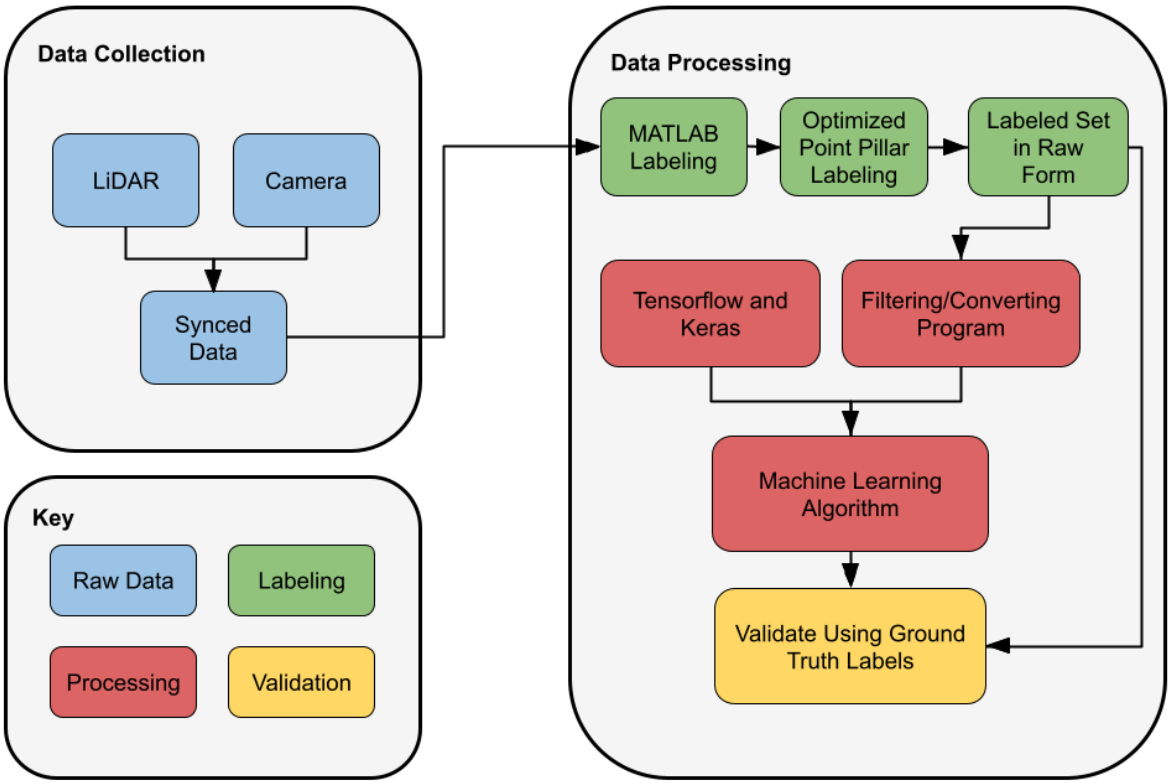

Figure 7: Second design iteration of the system.

By this point, we had been researching machine learning models for a while and understood that we should use Tensorflow and Keras instead of YOLO for compatibility reasons. We also looked into ways to label automatically, and discovered a promising network called Point Pillar that we hoped to use.

Another addition to this iteration was a filtering or converting layer we planned to create to allow multiple LiDAR formats. This was replaced by a combination of scripts we wrote in Python and Livox tools to convert files.

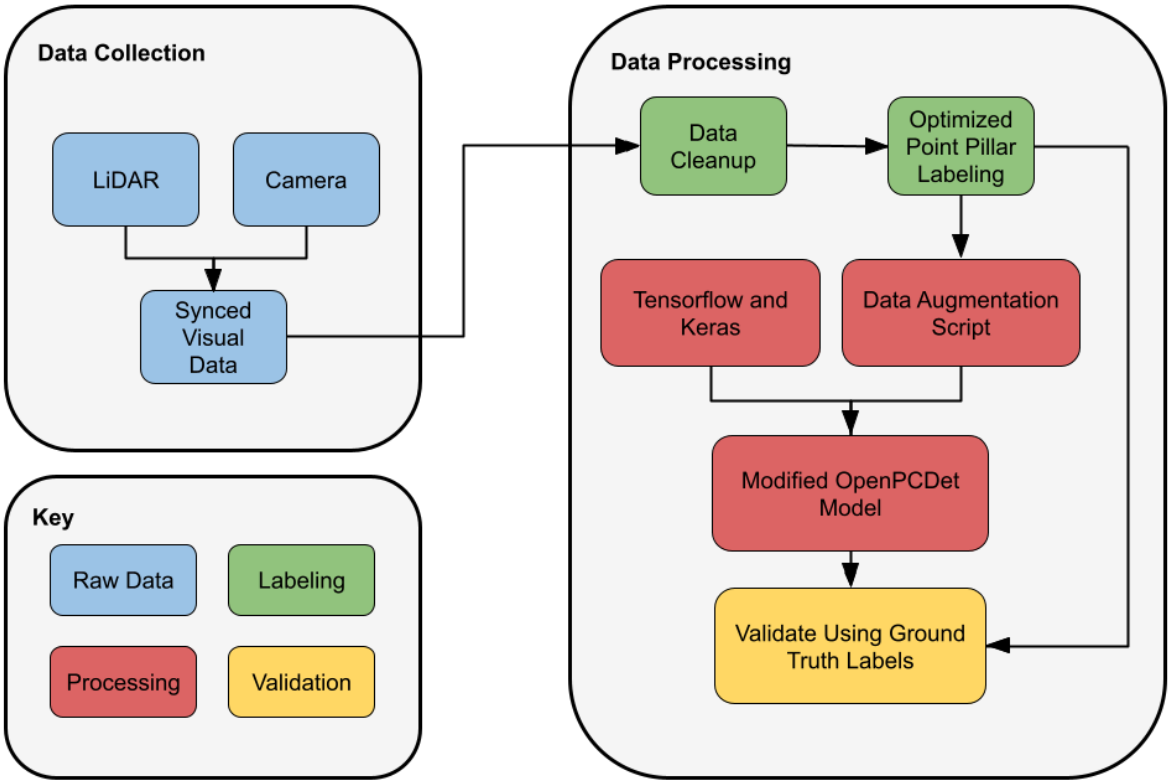

Figure 8: SE 492 initial design iteration of the system.

By this point in the implementation and design cycle, we knew we would use the OpenPCDet model and attempt to modify it. We kept most of the design the same as our final one; however, we still believed we could use Point Pillar in MATLAB to label automatically. We thoroughly researched it and attempted to use it during this semester, and discovered using it would require essentially training a new model, which would, in turn, need the labels we are trying to receive from the Point Pillars model.

Appendix 3 - Other Considerations

We learned a lot of lessons during our time working on this project. When we started, almost every group member knew nothing about machine learning and how to train or effectively gather data for a model. This caused us to spend a lot of time at the beginning of the project to try and get a baseline of knowledge to begin the project. One thing that we wished we could have done differently was to be more focused on our research and finalize our goal earlier in the process. We went on many tangents that were interesting but ultimately not as useful as we would have liked.

The project was exciting and engaging, which helped us push through the problematic troubleshooting issues surrounding the tools we were attempting to use for the project. At the beginning of the project, our client informed us that one of our challenges would be a lack of existing resources for the various problems we would encounter, and that was a lesson that we had reinforced early on and multiple times throughout the process. One of our biggest takeaways is to always expect there to be a complication that you were not aware of because this project had several.

All in all, we believe we chose a complex project and did it justice. We researched several implementations of tools we could use, and although many didn't work out, we used our new knowledge to discover new tools that we took advantage of. Even if we could not test our data at the level we wanted to, we made great strides in creating a pipeline of programs and tools to allow Livox LiDAR data to be used in a more competitive model than Livox's Detection Simulation repository.

Appendix 4 - Code

Python Label Conversion Script

After exporting the lidar labels from MATLAB in .csv format, converting them into a format acceptable to OpenPCDet is necessary. The Python code below converts the .csv exported from MATLAB into a directory of .txt files acceptable for OpenPCDet training. It can also print each label and save the labels in a .csv format.

```
import csv, shutil, os

dirPath = os.path.abspath(__file__).replace("\\", "/")[0:len(__file__)-len(os.path.basename(__file__))]

def getLabelInfo(title_row):
        '''
        Return label information from CSV

        :param title_row: List of strings in the first row of CSV file
        :return labels: names of labels (e.g. [Person, Vehicle]), labelRanges: Ranges of columns that
each label occupies in the CSV (e.g. [(1,195), (196, 450)])
        '''

        labelStartPositions = []
        labels = []
        title_row = title_row.strip()
        title_row = title_row.split(',')
        numOfDifLabels = (len(title_row) - 1) / 9
        label = ""
        for index, rawLabel in enumerate(title_row):
                if rawLabel != "Time":
                for char in rawLabel:
                        if not char.isnumeric() and not char == "_":
                        label += char

                if not label in set(labels):
                        labelStartPositions.append(index)
                        labels.append(label)
                label = ""

        labelRanges = getLabelColumnRange(labelStartPositions, numOfDifLabels)

        return labels, labelRanges

def getLabelColumnRange(labelStartPositions, numOfDifLabels):
        '''
        Returns the range of columns each label occupies in the CSV

        :param labelStartPositions: Starting position of each label (e.g. [1, 196])
        :param numOfDifLabels: Number of different labels in CSV title_row
        :return labelRanges: Ranges of columns that each label occupies in the CSV (e.g. [(1,195),
(196, 450)])
        '''

        labelRanges = []
        for index, value in enumerate(labelStartPositions):
        if index + 1 < len(labelStartPositions):
```

```
        labelRanges.append((value, labelStartPositions[index + 1] - value))
        else:
        labelRanges.append((value, int((numOfDifLabels * 9) - value) + value))
        return labelRanges

def getFilteredData():
        '''
        Filters out just the rows with labeled data from the CSV

        :return labelData: Labeled data from CSV in the format (frame_count, x, y, z, dx, dy, dz,
heading_angle, label_name)
        '''
        with open(dirPath + "exported_data.csv", newline='') as csvfile:
        rowIndex = 1
        colIndex = 1
        labels = []
        labelRanges = []
        labelData = []
        currLabelData = []
        title_row = csvfile.readline()
        labels, labelRanges = getLabelInfo(title_row)

        csvfile.seek(0, 0)
        reader = csvfile.readlines()

        for i in range(len(labelRanges)):
        for rowIndex, row in enumerate(reader):
                if rowIndex > 0:
                row = row.strip()
                row = row.split(',')
                if row[labelRanges[i][0]] != '':
                        for colIndex in range(len(row)):
                        if colIndex >= labelRanges[i][0] and colIndex <= labelRanges[i][1]:
                                if row[colIndex] != '':
                                currLabelData.append(row[colIndex])
                        currLabelData.append(labels[i])
                        currLabelData.insert(0, rowIndex-1)
                        labelData.append(currLabelData)
                        currLabelData = []

        return labelData

def parse_data(input_list, repeat_count):
        '''
        Parse out the frameCount, x, y, z, dx, dy, dz, heading_angle, label_name from each filtered row

        :param input_list: Filtered row from CSV
        :param repeat_count: Number of times data (x, y, z, dx, dy, dz, heading_angle) is repeated
        :return parsed_data: Data parsed into separate lists of (frameCount, x, y, z, dx, dy, dz,
heading_angle, label_name)
        '''
        parsed_data = []
        for i in range(1, repeat_count + 1):
        x = input_list[i]
        y = input_list[i + repeat_count]
        z = input_list[i + 2 * repeat_count]
        dx = input_list[i + 3 * repeat_count]
        dy = input_list[i + 4 * repeat_count]
        dz = input_list[i + 5 * repeat_count]
```

```
        heading_angle = input_list[i + 8 * repeat_count]
        label = input_list[len(input_list)-1]
        parsed_data.append([input_list[0], x, y, z, dx, dy, dz, heading_angle, label])
        return parsed_data

def printData(data):
        for i in data:
        print(i)

def saveToCSV(data):
        f = open(dirPath + "labeled_data.csv", "w", newline='')
        for row in data:
        writer = csv.writer(f)
        writer.writerow(row)
        f.close()

def saveToTextFiles(data):
        '''
        Save each frame of lidar data to a text file named after the frame number.

        :param data: Lidar data in an array [x, y, z, dx, dy, dz, heading_angle, label_name]
        '''
        if os.path.exists(dirPath + "labels/"):
        shutil.rmtree(dirPath + "labels/")
        os.mkdir(dirPath + "labels")
        if not os.path.exists(dirPath + "labels/"):
        os.mkdir(dirPath + "labels/")
        for row in data:
        f = open(dirPath + "labels/" + str(row[0]) + ".txt", "a")
        for index, item in enumerate(row):
        if index != 0:
                if index + 1 <= len(row):
                f.write(str(item) + " ")
                else:
                f.write(item)
        f.close()

def getLabeledData():
        filteredData = getFilteredData()
        resultData = []
        for val in filteredData:
        numOfLabelsInCurrFrame = int((len(val) - 2) / 9)
        for i in parse_data(val, numOfLabelsInCurrFrame):
        resultData.append(i)

        return resultData

def main():
        labeledData = getLabeledData()
        printData(labeledData)
        saveToCSV(labeledData)
        saveToTextFiles(labeledData)

main()
```

Conversion script from .las to .npy

This script converts our custom point cloud data into the format OpenPCDet expects. Points clouds are initially in .lvx format, which can be converted to .las using the file converter in Livox Viewer. Then, this Python script will convert the .las files to .npy.

```
import laspy
import numpy as np
import os

#change for linux
in_directory = r"C:\Users\sachp\sdmay24-31\OpenPCDet\data\livoxlas"
out_director = r"C:\Users\sachp\sdmay24-31\OpenPCDet\data\livoxnpy\points"

for file_name in os.listdir(in_directory):
    if(file_name.endswith('.las')):
        las = laspy.read(os.path.join(in_directory, file_name))

        x = las.x
        y = las.y
        z = las.z
        intensity = las.intensity

        np.save(out_director + '\\' + file_name[:file_name.index('.')] + ".npy", np.vstack((x, y, z,
intensity)).T)
```

Appendix 5 - Definitions and Utilized Tools

Deep Learning Model: A deep learning model is an artificial neural network with multiple layers (deep architecture) that enables automatic learning of hierarchical representations from data, facilitating the extraction of complex features and patterns.

OpenPCDet: OpenPCDet is a straightforward, simple, self-contained open-source project for LiDAR-based 3D object detection used in Livox Detection V2.0. This project utilizes a Point-Voxel Region-based Convolutional Neural Network (PV-RCNN)

LiDAR: a detection system that works on the principle of radar but uses light from a laser. It provides output data in the form of point clouds.

Point Cloud: A set of data points in a 3D coordinate system—commonly known as the XYZ axes. Each point represents a single spatial measurement on the object's surface. Taken together, a point cloud represents the entire external surface of an object.

Keras: Keras is an open-source high-level neural network API written in Python. It serves as an interface for building, training, and deploying artificial neural networks, simplifying the process of developing deep learning models. Keras is often used with other deep learning libraries, such as TensorFlow or Theano, and provides a user-friendly and modular approach to constructing neural networks.

TensorFlow: TensorFlow is a free open-source software library for machine learning and artificial intelligence. It can be used across various tasks but focuses on the training and inference of deep neural networks.

Livox Mid-40: A LiDAR sensor developed by Livox, a company specializing in LiDAR technology. The Livox Mid-40 LiDAR sensor is known for its compact design, high-performance capabilities, and cost-effectiveness. It is commonly used in various applications, including robotics, autonomous vehicles, and industrial automation, where precise and real-time 3D mapping is required.

Livox Viewer: Livox Viewer is a computer software designed for Livox LiDAR sensors and Livox Hub. Users can check real-time point cloud data of all the Livox LiDAR sensors connected to a computer and can easily view, record, and save the cloud data for offline or further use.

Cloud Compare: CloudCompare is a 3D point cloud (and triangular mesh) processing software. It was initially designed to compare two dense 3D point clouds (such as the ones acquired with a laser scanner) or between a point cloud and a triangular mesh. It relies on a specific octree structure dedicated to this task. Afterward, it has been extended to a more generic point cloud processing software, including many advanced algorithms (registration, resampling, color/normal/scalar fields handling, statistics computation, sensor

management, interactive or automatic segmentation, display enhancement, etc.). This tool will be explicitly utilized in our project to visualize the point cloud we gather from the Livox Mid 40.

Machine Learning: Machine learning is a subset of artificial intelligence that involves the development of algorithms and statistical models that enable computer systems to improve their performance on a specific task over time without being explicitly programmed. It relies on analyzing patterns and data to make predictions and decisions or identify trends, allowing machines to learn from experience and adapt to new information. Machine learning encompasses various techniques, including supervised learning, unsupervised learning, and reinforcement learning, and it finds applications in areas such as image and speech recognition, natural language processing, and predictive analytics.

MatLab: MATLAB, short for "Matrix Laboratory," is a high-level programming language and interactive environment primarily designed for numerical computing, data analysis, and visualization. It is widely used in academia, industry, and research for mathematical modeling, simulation, and algorithm development tasks. We used this to label our data using the LiDAR Labeler tool and the interpolation automation mode.

Lidar Toolbox: Lidar Toolbox provides algorithms, functions, and apps for designing, analyzing, and testing lidar processing systems on MATLAB. You can perform object detection and tracking, semantic segmentation, shape fitting, lidar registration, and obstacle detection. The toolbox provides workflows and an app for lidar-camera cross-calibration.

OpenPyLivox: Python3 driver for Livox lidar sensors

PyLas: Python library for lidar LAS/LAZ IO. LAS (and its compressed counterpart LAZ) is a popular format for lidar point clouds and entire waveform. Pylas reads and writes these formats and provides a Python API via Numpy Arrays.

ROS (Robot Operating System): A set of software libraries and tools that help you build robot applications

YOLO: "You Only Look Once" is a machine learning algorithm developed for real-time object detection. It operates with a single forward pass through a neural network, simultaneously predicting bounding boxes and class probabilities within a grid system. This makes it highly efficient for various applications such as surveillance, autonomous vehicles, and robotics.

Google Suite: This tool includes Google Drive, docs, slides, and more. We have utilized this to create detailed documentation updated in real-time for this project.

3D Detection & Tracking Viewer: This project was developed to view 3D object detection and tracking results. It supports rendering 3D bounding boxes as car models and rendering boxes on images.

Kitti Vision Benchmark Suite: Kitti contains a suite of vision tasks built using an autonomous driving platform. The complete benchmark contains many tasks, such as stereo, optical flow, visual odometry, etc. This dataset includes the object detection dataset, including the monocular images and bounding boxes. We are using Kitti to test the OpenPCDet model before using our data.

Livox ROS Driver: livox_ros_driver is a new ROS package specially used to connect LiDAR products produced by Livox. The driver can be run under Ubuntu 14.04/16.04/18.04 operating system with an installed ROS environment (indigo, kinetic, melodic). Tested hardware platforms that run livox_ros_driver include Intel x86 CPU and ARM64 hardware platforms (such as Nvidia TX2 / Xavier, etc.).

Linux Ubuntu 14.04/16.04/18.04: Operating system with ROS environment (indigo, kinetic, melodic) installed

Livox Detection: Combined with the advantages of HAP, this detector can achieve better perception performance than the Horizon. Another improvement is adopting an anchor-free method inspired by CenterPoint to make the detector more flexible in dealing with multiple datasets.

Livox Viewer: Livox Viewer is a computer software designed for Livox LiDAR sensors and Livox Hub. Users can check real-time point cloud data of all the Livox LiDAR sensors connected to a computer and can easily view, record, and save the cloud data for offline or further use. This is how we will initially visualize the data we gather before converting it.

Point Pillar: PointPillars is a method for 3-D object detection using 2-D convolutional layers. PointPillars network has a learnable encoder that uses PointNets to learn a representation of point clouds organized in pillars (vertical columns). The network then runs a 2-D convolutional neural network (CNN) to produce network predictions, decodes the projections, and generates 3-D bounding boxes for different object classes, such as cars, trucks, and pedestrians.

OBS: OBS Studio is a free, open-source, cross-platform screencasting and streaming app.