Granular classification of 3D Point Cloud objects in the context of Autonomous Driving

CMPE 255 Project Report

Deepak Talwar	Parshwa Gandhi	Hetavi Saraiya	Kedar Acharya
ID: 013744731	ID: 014518348	ID: 014508988	ID: 014151891

May 13, 2020

1 Abstract

The objective of this project is to use labeled 3D PointCloud data collected using LiDAR sensors on Autonomous Driving platforms to develop algorithms for classifying objects such as cars, pedestrians, trucks etc., as they appear in 3D PointCloud space. We attempt to use point cloud matching algorithms such as Iterative Closest point (ICP) [1], Iterative Closest Point Non Linear (ICP-NL) [2] and Normal Distribution Transform (NDT) [3] for the classification task. In addition, we also try using unsupervised classification methods such as K-Nearest Neighbors classifier [4] and compare the results. This technique may serve as a post object-detection step in an Autonomous Driving platform perception pipeline, wherein, it could help classify objects with finer granularity.

2 GitHub Repository

Please find our implementation at https://github.com/deepaktalwardt/point-cloud-clustering

3 Introduction and Motivation

To be able to successfully classify objects in 3D Point Cloud space, we first need to create models of how such objects "appear" to a LiDAR sensor in 3D Point Clouds space. Similar to how objects appear differently in camera images from different camera positions and orientations, objects appear differently in 3D Point Cloud space with different LiDAR sensor positions and orientations. This means that if the object is closer to a LiDAR sensor, it will reflect more points back to the sensor, whereas, if it is far away, it will reflect fewer points. Also, depending on the orientation, LiDAR might only be able to capture one side of the object as no laser beams will be hitting in the opposite side of the object. Thus, to create models for each object, we need to be able to see each object from multiple different viewpoints and extract the points that belong to these objects and concatenate them together in the same reference frame. Given enough instances, we will be able to create a dense cloud of points that make up a class of object as visible from a LiDAR.

This dense cloud of points can then be used for a variety of purposes. It can be re-input into other point clouds to augment datasets used for training deep-learning based models. It could be used to recreate mesh models in 3D reconstruction tasks, or it could be used to identify objects with finer granularity. The latter is the task we wish to address in this project. Normally, object detection algorithms only classify objects into broad classes such as "vehicle", "movable", "unnmovable", "pedestrian", etc. However, with this technique, we aim to provide an additional layer of classes. In this project, we attempt to further classify the "vehicle" class into "Sedan", "Jeep" etc., sub-classes.

4 Dataset Creation

For the purpose of this project, it is very important that we use object classes that appear consistently in the LiDAR space. Therefore, a simulated platform as a data source is a very good choice as we can limit the variety and number of types of objects that appear in the simulated world.

4.1 Data Collection

The simulation platform of our choice is the LGSVL Automotive Simulator [5]. This simulator provides LiDAR point cloud simulation, along with 3D object annotations which we need to extract objects from these point clouds. In addition, it has a tight integration with Robot Operating System (ROS) [6], which is useful for collecting, synchronizing and extracting data from the simulator. Figure 1 shows how the simulator looks with the LiDAR sensor visualization turned on.

Figure 1: LGSVL Simulator with simulated Point Clouds [5]

The entire process of data collection is shown in Figure 2. Each of the sections in this flowchart are detailed in the sections below.

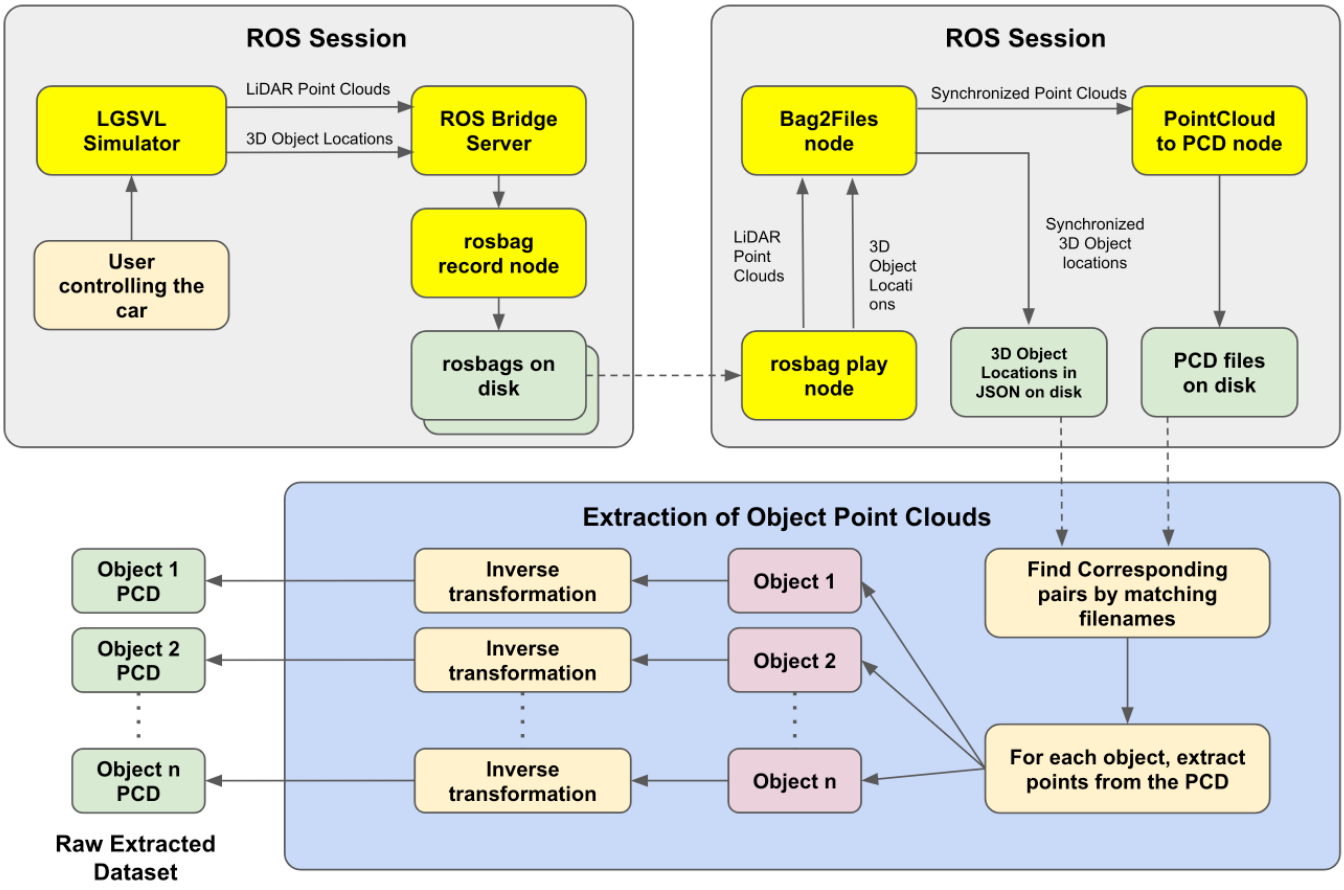

Figure 2: The entire flowchart of data collection into individual point cloud PCD and detections JSON files. This constitutes the raw extracted dataset that is ready to be processed.

4.1.1 Recording LiDAR Point Clouds and 3D Object locations to rosbags

The first step in the dataset creation process is to collect data into rosbags [7] from the simulator. Rosbags can be thought of as a video recording, except the frames in a video are replaced with LiDAR point clouds and their 3D annotations. Therefore, playing a rosbag is similar to playing back a recorded video. This is the best raw format to save simulated data into and is widely used in the robotics industry. This format allows us to have raw data available to us as it was streamed by the simulator, without having to run the simulator again.

To achieve this, we connected the simulator to a ROS Master Node through a ROS Bridge Server [8], launched a `rosbag record` node provided by the `rosbag` package [7] that would record all data being published by the simulator and store it in a `.bag` file. We drove the car inside the simulator to record data for several minutes. One such sample bag is available to be downloaded here:

https://drive.google.com/open?id=1YKRs-0sgGqqfWp0IsbvrfDyRIQixvVEG.

4.1.2 Synchronization and Extraction of files from rosbags

Now that the data is collected into rosbags, we need to extract the point clouds and 3D annotations from this bag. However, since the simulator publishes point clouds and 3D annotations asynchronously, we need to synchronize the point cloud and annotations time stamps. We set the

output frequency of the LiDAR point clouds and 3D annotations to the maximum frequency that is reliably output by the computer hardware we used. Running the simulator is computationally heavy for the GPU. After experimentation and trial and error, we achieved 27 Hz output frequency reliably with NVIDIA RTX 2080 GPU. Then, using the `ApproximateTimeSynchronizer` class provided by `message_filters` [9] package, we synchronize the LiDAR point clouds and 3D annotations, such that, they were published within 10 ms of each other. This ensures that we are keeping snapshots of only the instants in time that we have synchronized. Doing this means that we drop around 60 % of our original data in the rosbag.

Furthermore, we also drop the LiDAR point clouds that are scanned when there are no objects of interest in the frame. This is so that we only account for the LiDAR scans that have at least one annotation (meaning that there is at least one object of interest nearby) associated to them.

Once we have achieved the synchronization, we save the point clouds into `.pcd` files and the annotations in `.json` files so that they can be used later. To easily associate `.pcd` and `.json` files, we name them both with the timestamp of the LiDAR point cloud message. This maintains the correspondence while providing uniqueness in file names. An example of such JSON file is shown below:


```
{
    "header": {
        "stamp": {
            "secs": 1586102742,
            "nsecs": 765440
        },
        "frame_id": "",
        "seq": 375
    },
    "detections": [
        {
            "header": {
                "stamp": {
                    "secs": 0,
                    "nsecs": 0
                },
                "frame_id": "",
                "seq": 0
            },
            "score": 1.0,
            "bbox": {
                "position": {
                    "position": {
                        "y": 0.76051902771,
                        "x": -10.9455060959,
                        "z": -1.98804783821
                    },
                    "orientation": {
                        "y": 0.000263146037469,
                        "x": 0.00144024332985,
                        "z": 0.00534463580698,
                        "w": -0.999984622002
                    }
                },
                "size": {
                    "y": 2.0824213028,
                    "x": 4.56552648544,
                    "z": 1.35600721836
                }
            },
            "velocity": {
                "linear": {
                    "y": 0.0,
                    "x": 9.62840080261,
                    "z": 0.0
                },
```

```
                "angular": {
                    "y": 0.0,
                    "x": 0.0,
                    "z": 0.235796287656
                }
            },
            "id": 136,
            "label": "Sedan"
        }
    ]
}
```


From the JSON object, you can see that the `header` contains the timestamp and the `detections` list contains all the objects detected at this time. The `bbox` element contains the location, orientation and the size of the bounding box of this object, and the `label` tells us that this object is a "Sedan".

The corresponding point cloud from this time when saved in PCD format looks like the following in a text editor:

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z intensity timestamp
SIZE 4 4 4 1 8
TYPE F F F U F
COUNT 1 1 1 1 1
WIDTH 19954
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 19954
DATA ascii
1.674962 0.0019691722 -2.3125429 0 0
1.6718343 -0.066040121 -2.3111014 0 0
1.6722711 -0.13412777 -2.3123593 0 0
1.6685519 -0.20209752 -2.3119102 0 0
1.6645238 -0.27005503 -2.3120055 0 0
1.6563416 -0.33765778 -2.3103778 0 0
1.6517398 -0.40552381 -2.3114486 0 0
...
...
...
```

As apparent from the PCD file above, this cloud contains x, y, z fields. Although it has allocated space for *intensity* and *timestamp* fields, they are populated with zeros by the simulator. This cloud contains 19954 points and it is an "unorganized" cloud, meaning that it is arranged as a vector, instead of as an array. This is apparent from the "HEIGHT" value equal to 1. These x, y, z values are defined in the LiDAR reference frame. For instance, a point $(1.67, -0.13, -2.31)$ is 1.67 m ahead of the LiDAR scanner, 0.13 m to the right of it, and 2.31 m below it. When viewed in the PCD Viewer tool [10], we get the following rendering:

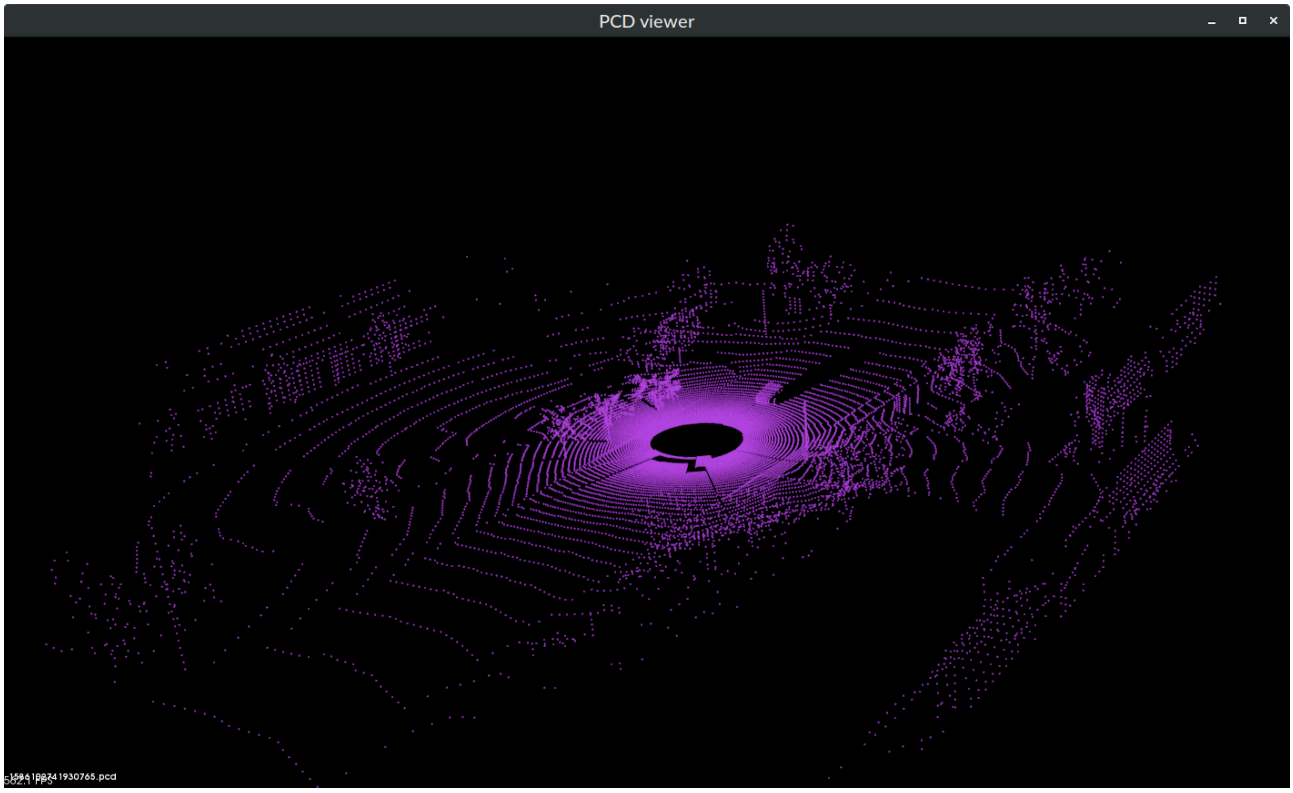

Figure 3: PCD Viewer displaying the entire point cloud

The code for saving ROS bags to PCD and JSON files is provided as a ROS Python node, `bag2files.py`, here https://github.com/deepaktalwardt/point-cloud-clustering/blob/master/ros_ws/src/dataset_generation/scripts/bag2files.py. In addition, we employ the `pointcloud_to_pcd` node provided in the `pcl_ros` [11] package to save the synchronized point clouds as PCD files.

4.2 Data Preprocessing

With the data extracted into individual PCD-JSON pairs, we now need to further process the data before we can use it in any kind of data analysis techniques.

4.2.1 Extraction of Object Point Clouds

The first step, is to extract individual objects from these point clouds and save them into individual `.pcd` files. This is important because then we will be able to combine all instances of the same object into a single point cloud which will serve as our model for that object type.

This is done by using the Point Cloud Library (PCL) [12] in C++. First, corresponding PCD-JSON files are found, then the detections in the JSON file are parsed. For each detection, we create a `pcl::CropBox<pcl::PointXYZ>` type object and set its location, size and orientation as described in the JSON file (see 4.1.2). This crop box is then extracted from the point cloud, inverse transformation back to the world reference frame is applied to all points within this crop box (see 4.2.2 for details) and then, saved as an individual `.pcd` file.

At this step, we can specify N_{thresh}, the minimum number of points that an extracted point cloud must contain in order for it to be saved as a `.pcd` file. This threshold plays an important role in the classification problem ahead. We repeat this process for all detections in all point

clouds. This gives us all objects as individual point clouds. We name the files with the following convention:

`<class>-<name of source PCD file>-<index of this class in source PCD file>.pcd`.

Example: `Hatchback-1589078382118914-2.pcd`.

This ensures uniqueness, traceability to the source PCD file, and gives us an easy way to know the class of the object from its name. Example of an extracted "Sedan" point cloud is shown in Figure 5.

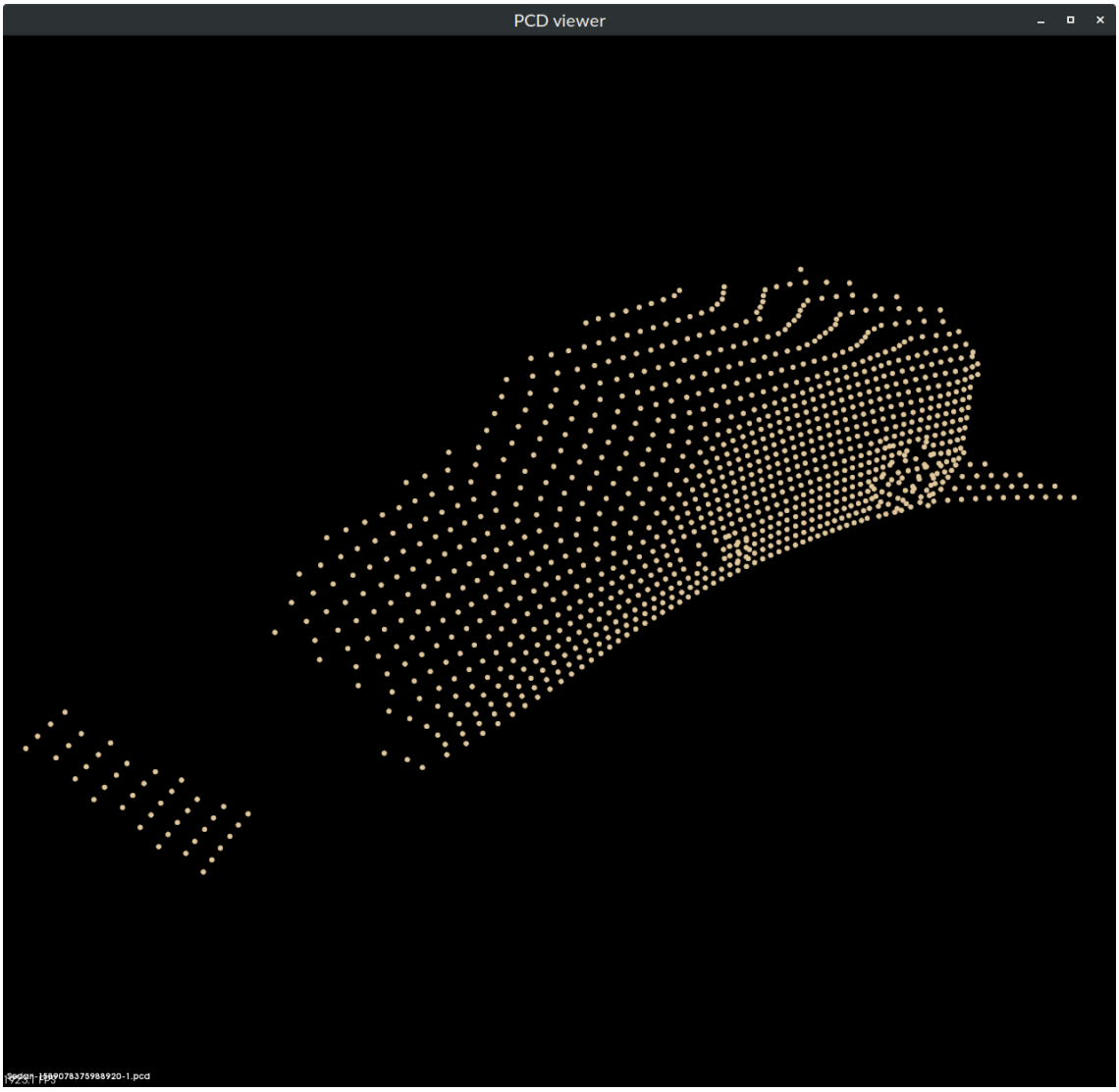

Figure 4: PCD Viewer with a single "Sedan" object. Note that in this view, the LiDAR sensor can only "see" the left-side of the car, which is why there are no points on the right-side or rear of the car, thus, necessitating the need for combining multiple clouds to make a complete model.

The code to achieve this is provided in the following two files:

1. Implementation of all functions needed for point cloud extraction: https://github.com/deepaktalwardt/point-cloud-clustering/blob/master/ros_ws/src/dataset_generation/include/dataset_generation/extract_point_cloud_objects.h

2. The file with the main function to perform the extraction: https://github.com/deepaktalwardt/point-cloud-clustering/blob/master/ros_ws/src/dataset_generation/src/extract_point_cloud_objects_node.cpp

4.2.2 Inverse transformation

Before the object point clouds can be saved to PCD files, there is an inverse transformation step that we need to perform. Since our goal is to create a full model of the objects as they appear to a LiDAR sensor, we need to be able to concatenate multiple clouds of the same class with each other. However, as these point clouds stand right now, they cannot be concatenated together to form the model as these points are referenced to their original inertial frames of references that they were initially extracted from. To concatenate them, we need to transform these points (i.e., apply a translation and rotation to these points) such that they are all referenced from the same origin.

The easiest way to do this is to apply the inverse of the transformation that was used to extract the crop boxes in the first place. After applying this transfomation, these points will be referenced from global origin 0, 0, 0 and rotated to the same orientation (x forward, y left, and z up aka right-hand reference frame) and thus can simply be concatenated together. This is done using the `pcl::transformPointCloud()` [13] functionality provided by PCL and using the inverse of the locations and orientations of the detections.

4.2.3 Visualization of extraction

To ensure that we are extracting the correct points from the point clouds visually, we created a tool that draws the previously mentioned cropboxes on to the point cloud. This is needed because if our point cloud extraction is flawed, all the steps thereafter will produce bad results. This visualization showed us some issues with our extraction which we needed to fix. Figure 5 shows the visual of this tool.

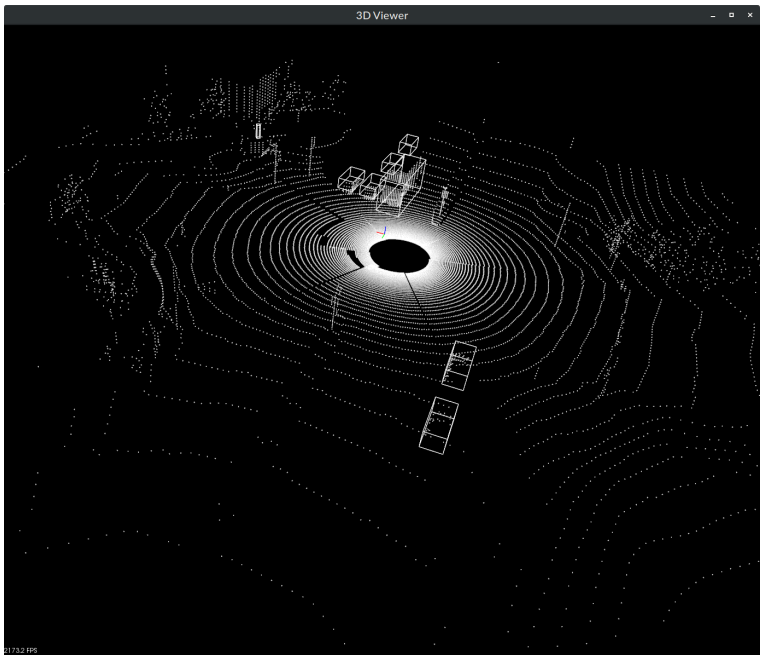

Figure 5: Visualization tool that draws boxes around the objects

The code for this visualization tool is provided here: https://github.com/deepaktalwardt/point-cloud-clustering/blob/feature/deepak/visualize-cropboxes/ros_ws/src/dataset_generation/include/dataset_generation/extract_point_cloud_objects.h

5 Creation of LiDAR Point Cloud models

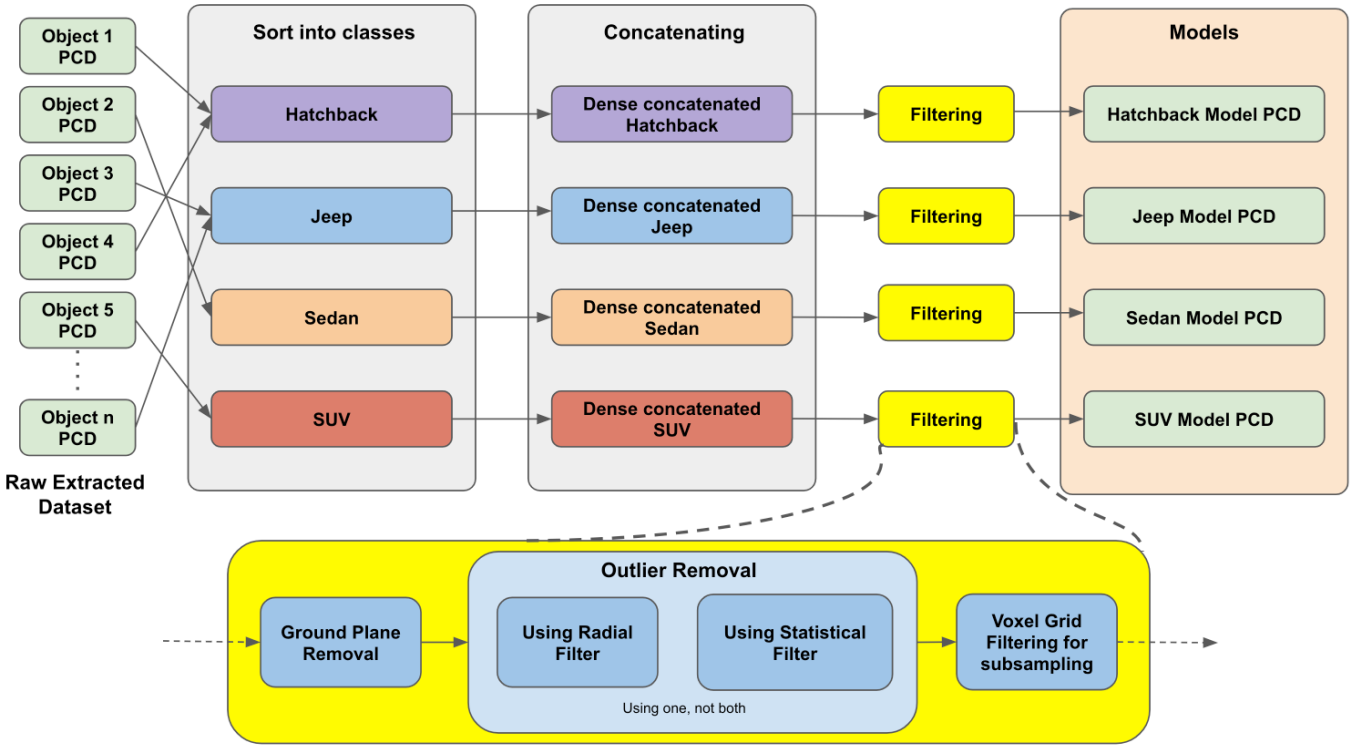

Figure 6: This flowchart shows the entire process for LiDAR Point Cloud model creation. It also shows the steps employed for Filtering the models.

With all object point clouds saved as individual PCD files, we now begin the process of creating our combined models. Figure 6 shows the entire process of creation of the models. This prcocess is detailed in the following sections.

5.1 Classes of objects

The first step in this process is to define the classes of our dataset. The LGSVL Simulator [5], by default, supports seven classes of objects - Sedan, SUV, Hatchback, BoxTruck, SchoolBus, Pedestrian, Jeep. Out of these classes, we choose to use four classes, namely - Sedan, SUV, Hatchback and Jeep. There are two main reasons for this choice:

- Pedestrian point clouds generally contained very few points (< 10), which made creating models really difficult. Additionally, pedestrian models are not "rigid" and simply combining them leads to very noisy models.
- BoxTruck and SchoolBus classes seem to spawn much less frequently within the simulator, which resulted in far fewer clouds for these classes.

Given these classes, we then set $N_{thresh} = 10$, and extracted all object clouds for these classes as described in 4. Since these names of these files contains the class they belong to, we can sort them when we load them from disk.

5.2 Concatenation of Point Clouds

Now, to get a complete model of each of our classes, we can begin concatenating the object point clouds that belong to the same class. The reason why this is possible is because we inverse transformed these objects into the same reference frame, as explained in 4.2.2. We tried two different approaches for this step.

5.2.1 Simple Concatenation

In this approach, we iterate over all object point clouds in the same class and simply accumulate those points into a combined point cloud. Given that these objects belong to the same class and are now defined in the same reference frame, they should theoretically match and align perfectly. However, this is not what we observe after concatenation. This is because of the asynchronous output of the LiDAR Point clouds and 3D annotations from the simulator, as explained in 4. Due to the maximum 10 ms difference in recording time of the Point Cloud and the annotation, the object in the cloud is not always perfectly centered. This displacement from the center is directly proportional to the relative velocity between the user car (that has the LiDAR sensor) and the object vehicle. If the relative velocity is large, the displacement will be larger, and vice versa.

Figure 7 shows the unfiltered concatenated model for the "Jeep" class. As apparent from the image, all sides of the vehicle are visible when the clouds are concatenated. However, the surface of the jeep is not sharp because of the relative movement mentioned above. In addition, there are many points collected from the road surface which are outliers and must be removed, as well as many flying points that add noise. The filtering process is explained in 5.3.

5.2.2 Concatenation with Iterative Closest Point [1]

To mitigate the mismatch problem that arises in the process of simple concatenation, we attempted to perform a different technique to concatenate in order to achieve sharper surfaces on the objects. In this technique, we use the Iterative Closest Point algorithm [1] during the process of concatenation to first align the clouds before concatenating them together. This would eliminate the mismatch that appears due to the asynchronous output of LiDAR point clouds and 3D annotations and result in a much better model.

To do this, we first manually created initial source models for ICP. These were created by handpicking a few object clouds for each class and concatenating them using simple concatentation, such that, the object was completely visible from all sides. Then, we iterated through all the clouds of that class, used ICP for aligning them to the source, and if ICP converged, we would concatenate them. This resulted in sharper looking models with less noise.

However, after filtering and preprocessing, the quality of the models concatenated with or without ICP was comparable. As a result, we chose to simply use the filtered models that were concatenated without the use of ICP.

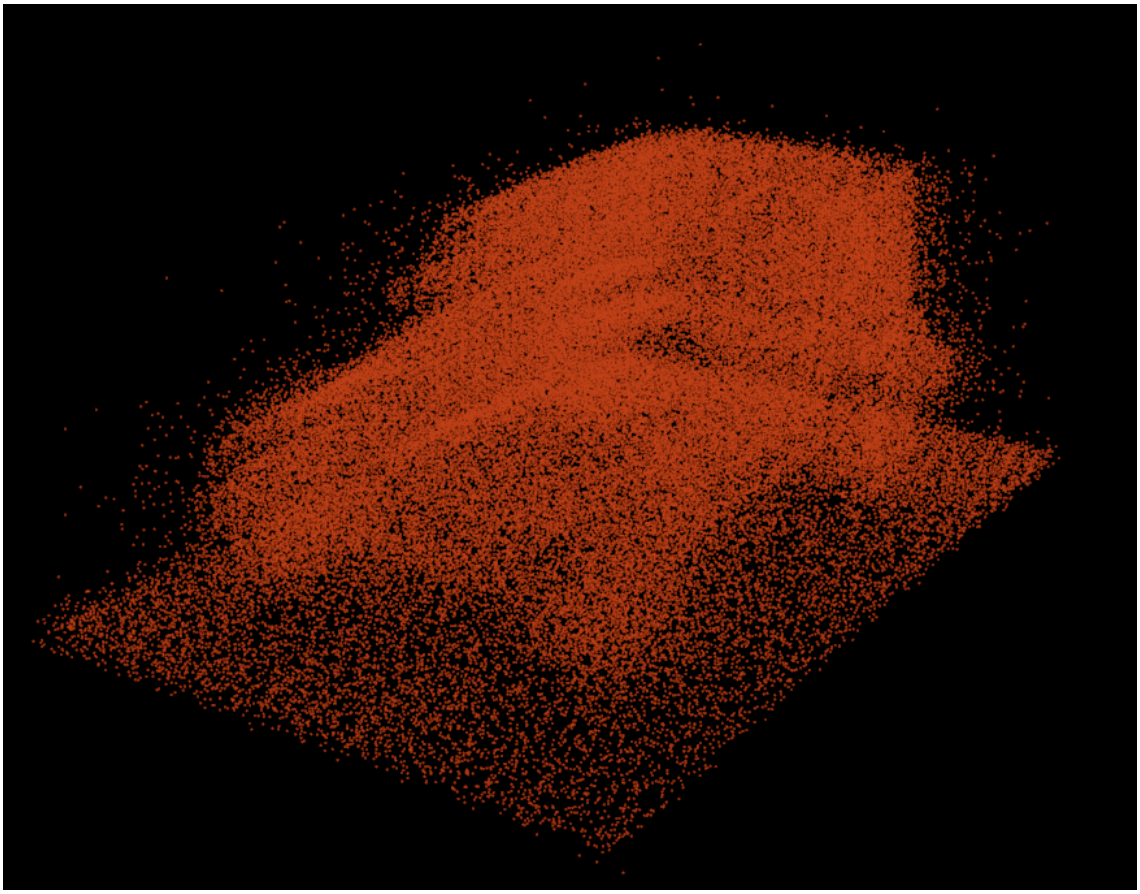

Figure 7: Simple concatenated model of the "Jeep" with 95612 points. This model is unfiltered and contains many outliers.

5.3 Data Cleaning and Filtering

Sections 5.2.1 and 5.2.2 demonstrate the two ways of concatentation that we used for creating our models. However, the output of these models need to be filtered and cleaned before they can be used and have the following main issues:

1. Models contain many outlier points. In terms of noise, floating points and points on the ground.

2. Models contain too many points for fast convergence with matching algorithms.

3. There is a big mismatch between the number of points of individual scans and models.

To address these issues, we created a filtering pipeline with three major steps that are described in the sections below. Figure 6 shows the various stages involved in the filtering process. Figure 8 shows the outputs of the intermediate stages of the filtering process for the "Jeep" model. Similar techniques were used for creating models for all other classes as well. The parameters for the different stages of filtering were chosen iteratively after trial and error.

The implementation of the following filtering techniques, along with visualization and saving functions are provided in the `PointCloudProcessing` class here: https://github.com/deepaktalwardt/point-cloud-clustering/blob/master/ros_ws/src/dataset_generation/include/dataset_generation/process_point_clouds.h

Figure 9 shows how all the final models look like after appropriate filtering and cleaning has been performed.

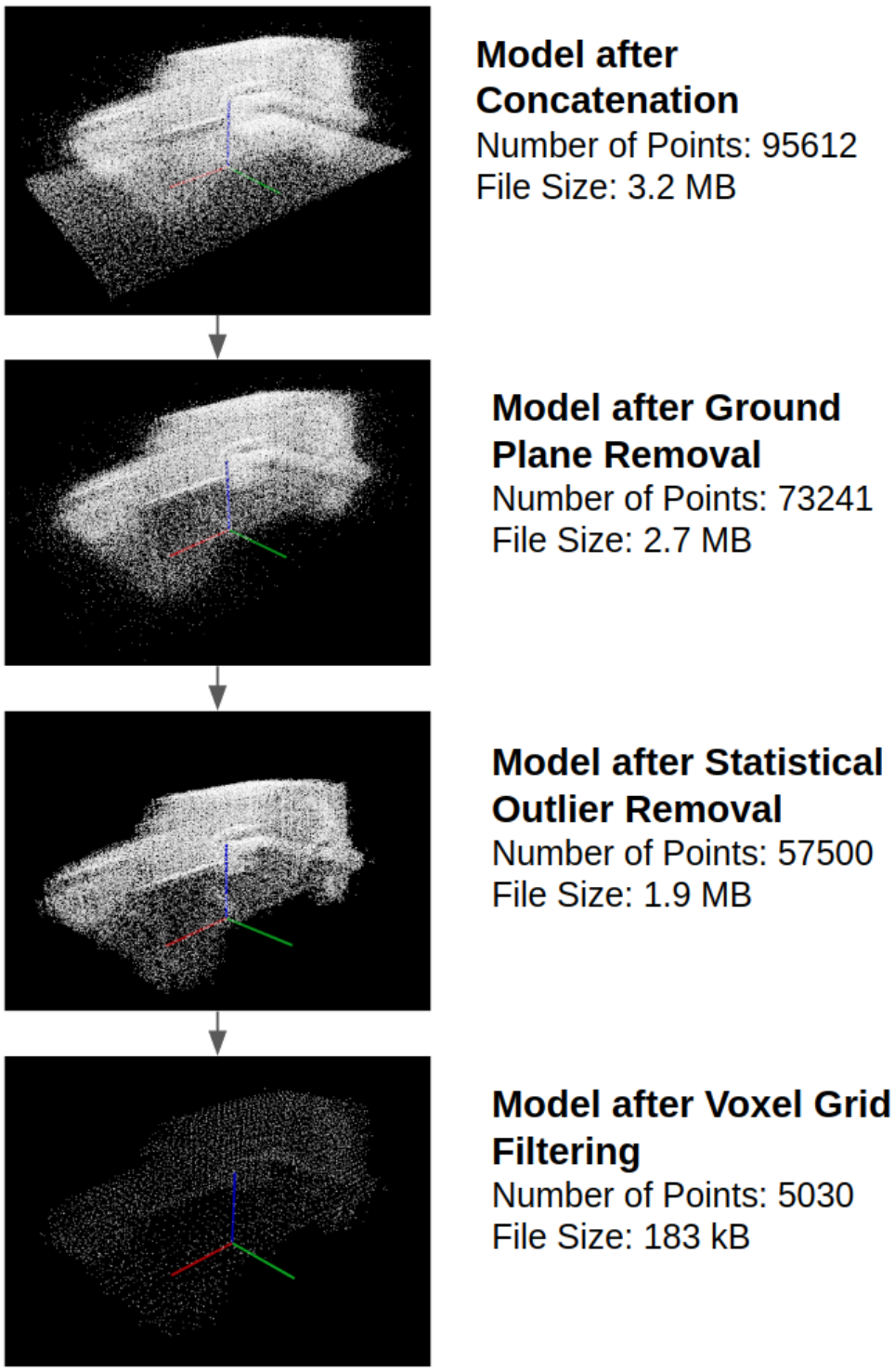

Figure 8: Shows the outputs of the intermediate filtering processes. The first image shows the raw model after concatentation. Second image shows the output after ground plane removal. Third image shows the output after statistical outlier removal and fourth image shows the output of Voxel Grid filtering, which is then saved as the model to be used.

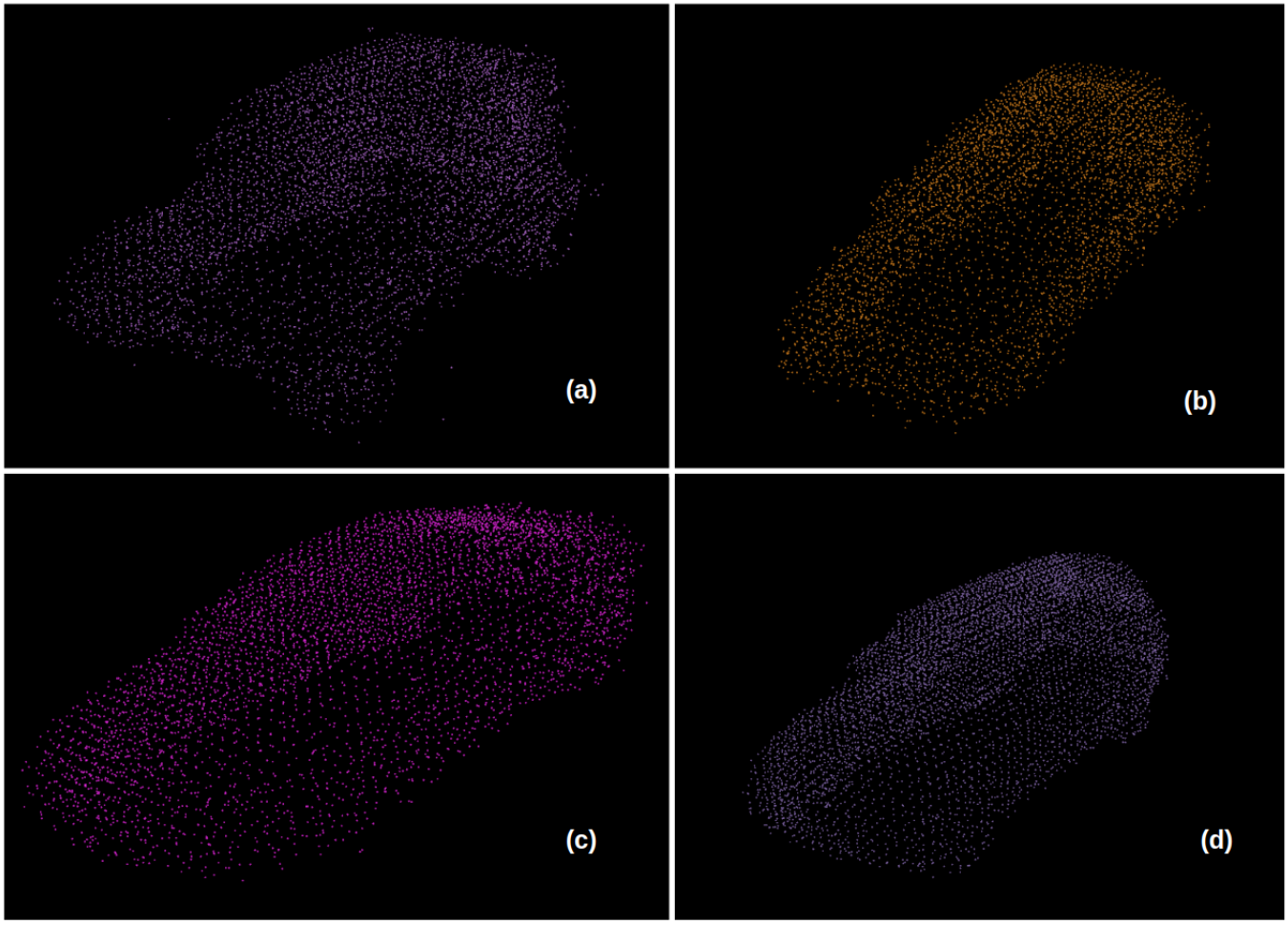

Figure 9: (a) Model of Jeep, (b) Model of Hatchback, (c) Model of Sedan, (d) Model of SUV

5.3.1 Ground Removal using PassThrough Filtering

To create accurate models of the objects, we treated the points that were reflected from the road surface as outliers and removed them using pass through filter. The PCL Library [12] provides the `pcl::Passthough<pcl::PointXYZ>`[14] class which works well for this purpose. This class takes in a field name (in our case x, y, z) and a minimum and maximum value, and only retains the points that lie within that range of values.

In our case, after some experimentation, we found that vehicle bodies are within the range of 0.2 m to 2 m along the z axis. As a result, setting `setFilterFieldName` to z and min and max ranges to 0.2 and 2 respectively, removed all the points that lie on the ground.

5.3.2 Outlier Removal using Radial Filtering

This method of outlier removal removes points from a point cloud that do not have a given number of neighbors within a specific radius from their location. Radial outlier removal uses the distance (radius) and number of neighborhood threshold. Points that do not satisfy this threshold are detected as outliers. A `pcl::RadiusOutlierRemoval<pcl::PointXYZ>` [15] class's object has `setInputCloud(inCloud)`, `setRadiusSearch(Radius)`, `setMinNeighborsInRadius(MinNeighbor)`

methods to set parameters. For filtering, the `filter()` method is used. This method is useful for removing points that are far away from a cluster of points, however, it does not work very well if there is a cluster of outliers outside the main object.

5.3.3 Outlier Removal using Statistical Filtering

The PCL library [12] provides a `pcl::StatisticalOutlierRemoval<pcl::PointXYZ>` class [16] that identifies outliers in a point cloud by performing statistical analysis on each point's neighborhood, and then trimming those points that do not follow a criterion. For each point, the `pcl::StatisticalOutlierRemoval<pcl::PointXYZ>` class [16] object computes the mean distance, μ, to K number of its neighbors. Also, it assumes that the distribution of points in the neighborhood of each point is gaussian and thus also calculates the standard deviation σ of this neighborhood. It then trims all the points that lie m standard deviations away from the mean, where, m is a multiplier to the standard deviation.

The parameters K and m are inputs to this filter and from our experimentation, values of $K = 5, m = 0.3$ worked well in removing the outliers. Figure 8 shows the output of applying this filter.

5.3.4 Voxel Grid Filtering for subsampling

In three dimensional graphics, the volume is divided into evenly spaced columns and rows in all the three directions. These cubes are called voxels. A voxel represents a value on a regular grid in three dimensional space. Our dataset is made up of concatenated point clouds where the total number of points in an unfiltered model can sum up to 100,000. In order to down sample this data, we have used Voxel Grid Sampling. Voxel Grid sub-samples the data by taking the spatial average of the points in the clouds. All the points will be down sampled with respect to centroid of the voxel. With the help of this filter we reduce the size of the point clouds and bring the number of points down to roughly 5,000.

The PCL library provides `pcl::Voxel_Grid<pcl::PCLPointCloud2>` [17] class which down samples the point clouds. An object of this class assembles a local 3D grid over a given point cloud and filters it with the leaf size on each axis. The leaf size from our experimentation for all the 3 axis are 0.1.

We attempted to subsample all our models such that they were roughly of the same size, with the same number (roughly 5,000) of points.

6 Alternative model representation: Mesh models

We experimented with representing our clean point cloud models as mesh surface models instead of points only. This is done by estimating the normals of local surfaces formed by neighborhood points, which act as vertices for the surfaces. With some experimentation, we achieved subpar results as shown in Figure 10. As a result, we decided not to use this representation for the remainder of the analysis. It appears that the clouds need to be a lot cleaner for the local normal-estimation to work correctly.

The implementation of this conversion is also provided in the `PointCloudProcessing` class here: https://github.com/deepaktalwardt/point-cloud-clustering/blob/master/ros_ws/src/dataset_generation/include/dataset_generation/process_point_clouds.h

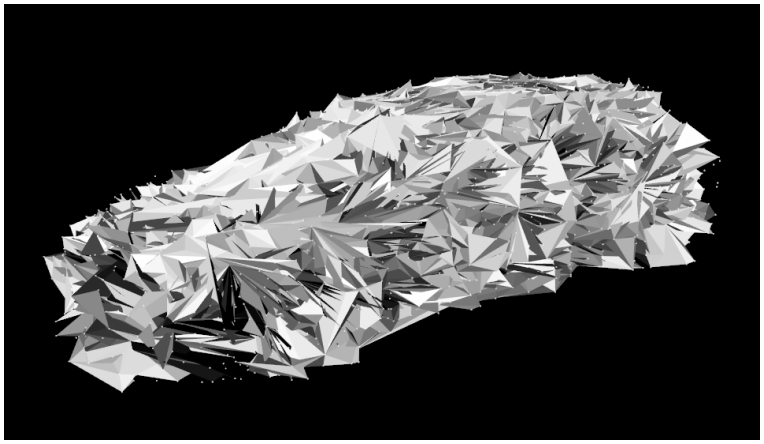

Figure 10: Shows the "Hatchback" model represented as a mesh.

7 Test dataset

To create the test dataset for this project, we collected a completely new rosbag from the LGSVL simulator [5] by driving around in the same virtual environment. We extracted the object files from this dataset using the same process as explained in 4.

We created several different testsets from this rosbag by changing the value of N_{thresh}. With $N_{thresh} = 10$ the test dataset contains 4 classes and the number of objects in the test set is as below:

- Hatchback : 1559
- SUV : 1369
- Sedan : 2062
- Jeep : 2384

The total number of objects adds up to 7374. Additionally, we created testsets with $N_{thresh} \in 10, 50, 100, 300, 500$ which we use later for testing.

Figure 11 shows the distribution of the number of points per object point cloud for each class. As visible from the plots, most object point clouds have < 50 points, therefore, it is interesting to see how our algorithms perform as we vary N_{thresh}.

8 Classification of Testset

With the dataset extraction procedure determined, and the test dataset defined, we can now introduce the various algorithms that we used for solving the point cloud classification problem. We broadly used two main category of methods:

- Point cloud matching (or registration) based methods
  - Iterative Closest Point (ICP) [1]
  - Iterative Closest Point Non Linear (ICP-NL) [2]
  - Normal Distributions Transform (NDT) [3]

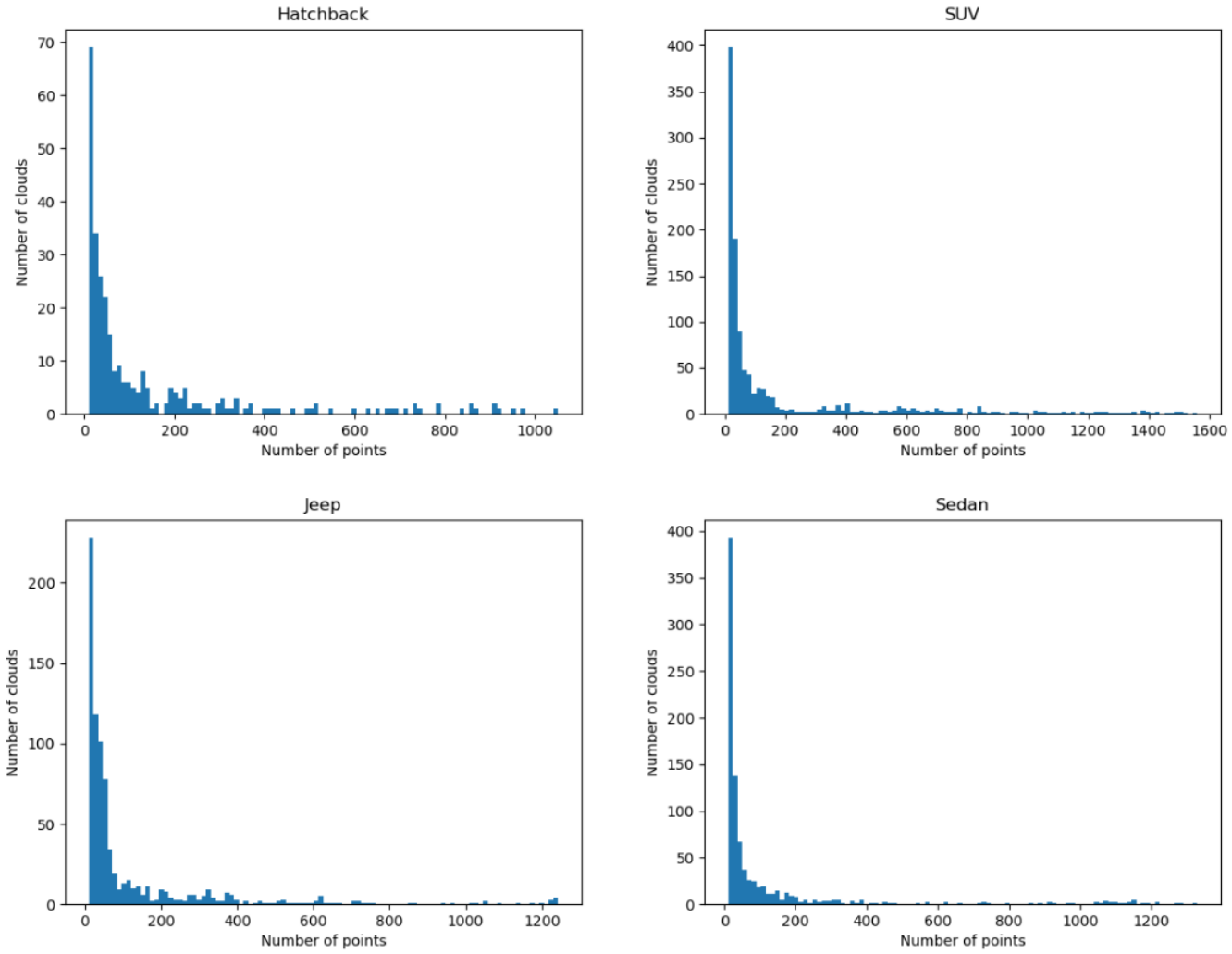

Figure 11: Shows the distribution of the number of points per object point cloud for all classes

- Unsupervised Classification algorithms
  - K-Nearest neighbor Classification [4]

The following sections describe these algorithms in detail and demonstrate the results we achieved.

8.1 Point Cloud matching: Iterative Closest Point [1]

Iterative Closest Point (ICP) is one of the most widely used point cloud matching algorithms. This algorithm is mainly used for iteratively aligning point clouds together such that the euclidean distance between their corresponding points gets minimized. Thus, it iteratively solves an optimization problem. ICP uses the Singular Value Decomposition (SVD) to solve the optimization problem. The return value for ICP between source and target point clouds is a rigid-transformation matrix, which encodes a rotation matrix as well as a translation vector, which when applied to the source point cloud, matches it with the target point cloud as closely as possible.

For the purposes of classification, however, we are not concerned with the output rigid-transformation. Instead, we want to use the quality of the transformation as a metric to predict

the class of the object. For this purpose we use two different metrics.

- Convergence: The algorithm is said to have converged when there is not a significant change in the rigid-transform in-between iterations. This means that the matching is as good as it can be.

- Fitness Score: This is the mean euclidean distance between the corresponding keypoints after the algorithm has finished running. The lower the fitness score, the better the alignment.

Thus, to classify using ICP, we iterate over all test point clouds and attempt matching them with all four models of classes (as defined in 5.1) and then use the above mentioned metrics to predict the point cloud's class.

For the purposes of this project, we use these metrics to build three different testing criteria:

- Criterion 1: Consider only the point clouds that converged to a single model. In this case, the class predicted is the one to which the cloud converged to.

- Criterion 2: Consider the point clouds that converged to one or more models. In this case, the class predicted is the one to which the cloud converged to, and has the lowest fitness score.

- Criterion 3: Consider all point clouds regardless of convergence. In this case, the class predicted is the one which has the lowest fitness score.

At this point, it is helpful to demonstrate what ICP convergence looks like. Figure 12 shows attempting to match a Sedan cloud (in white) with the model of the Jeep (in red) after 1 iteration. As clearly visible from this figure, the Sedan does not match with the model of the Jeep and thus the fitness score will be high.

On the other hand, figure 13 shows successful matching of a Jeep cloud (in white) with the model of the Jeep (in red) after 1 iteration. In this case the fitness score will be low and thus the algorithm will predict Jeep as its class.

8.1.1 Results

In this section, we provide results of the classification using ICP. We experimented with many different settings with ICP and settled to the following parameters:

1. Maximum number of iterations: 1

    - Although ICP requires multiple iterations to provide good matching rigid-transformation, we found that limiting the max number of iterations to 1 gives us the best classification result. Additionally, as mentioned above, we do not use the rigid-body transformation. Also, since the original alignment of the test clouds and model clouds is already pretty good, a single iteration of ICP should make it better.

2. Maximum Correspondence Distance: 0.01 m

    - This is dependent on the linear resolution of the LiDAR sensors provided by the LGSVL Simulator [5].

3. Euclidean Fitness Epsilon: 0.001

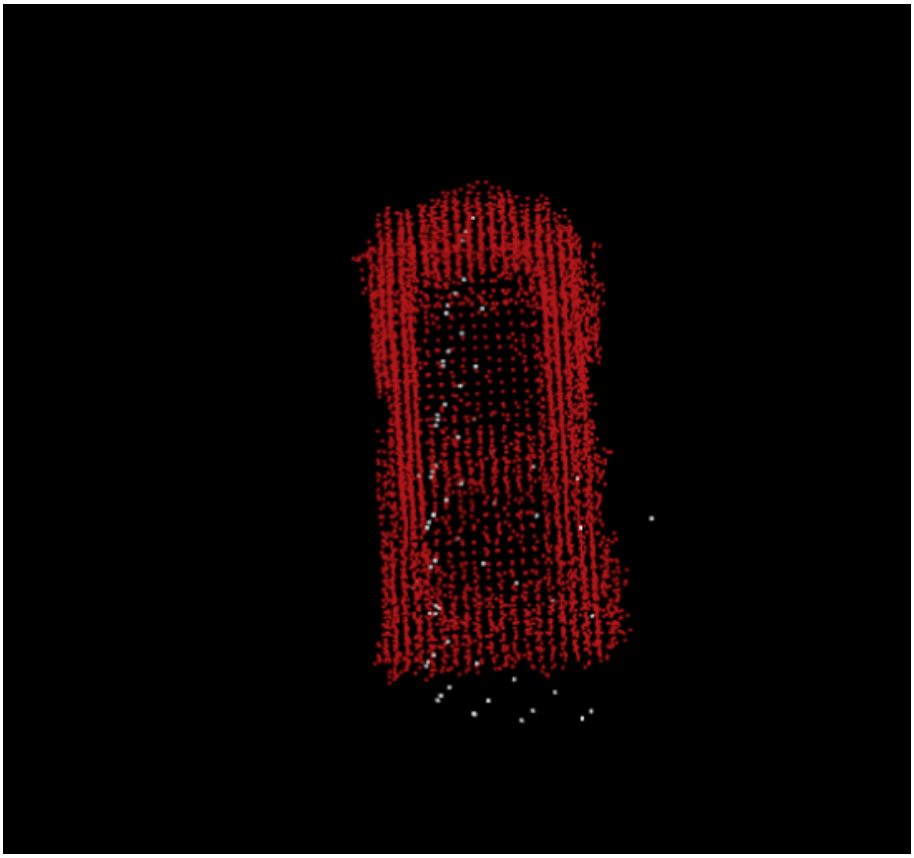

Figure 12: Attempting to match a Sedan (white) point cloud with a Jeep (red) point cloud model, after 1 iteration.

- This is amount of reduction in fitness score needed, otherwise it is considered to have converged.

In this section, we only present the results of using Criterion 2 for classification. While using Criterion 1 for classification yielded better results, the downside is that we do not obtain a prediction for point clouds that do not converge to only one of the models after a single iteration. On the other hand, using Criterion 3 provides us results for all point clouds, however, classification results are poor for point clouds that do not converge.

Figure 14 shows the confusion matrix obtained using ICP classification on the testset with $N_{thresh} = 300$. As apparent from the matrix, the classification works quite well for Jeep and SUV classes, however, it doesn't work very well with Hatchback and Sedan classes. This is very likely because Hatchback and Sedan models looks very similar as shown in figure 9. The big difference between the Hatchback and Sedan models is in the rear, where the Hatchback is flat, whereas, the Sedan is elongated. Therefore, all the point cloud objects that only "see" the front, could match with either of the classes, whereas, point cloud objects that "see" the rear will most likely align correctly. Confusion matrices for ICP Classifications on other N_{thresh} value testsets are provided in Appendix A.

Figure 15 shows the precision and recall values for ICP Classification over testsets with different N_{thresh} values. Surprisingly, there is not a significant change in these values depending on N_{thresh}, which means that all the clouds that do not converge to any of the models are common in all of these testsets.

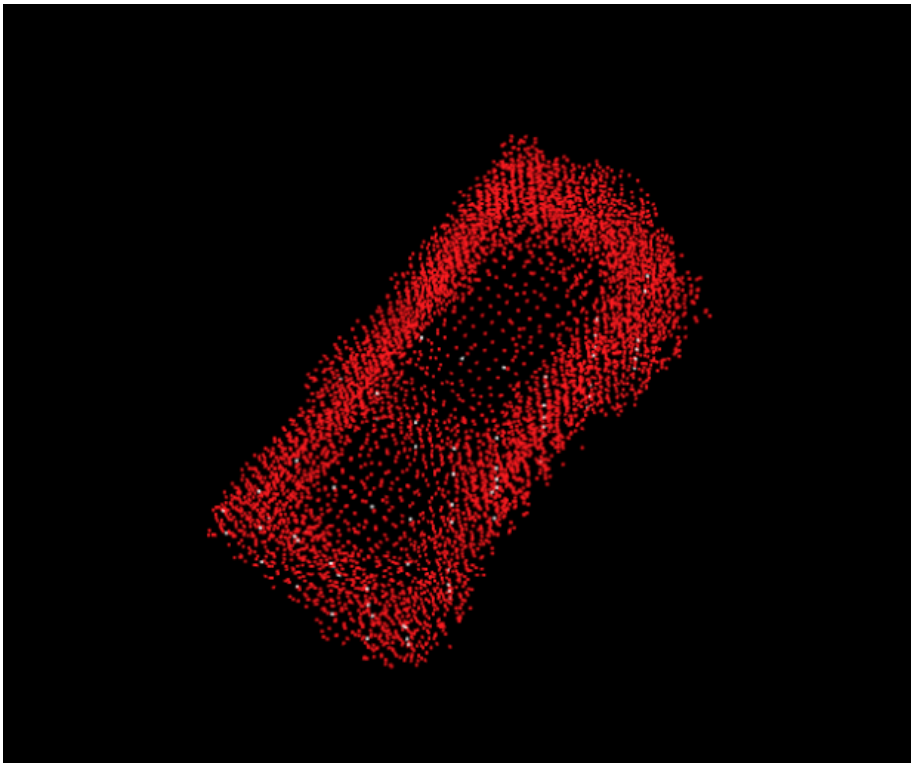

Figure 13: Successful matching of a Jeep (white) point cloud with a Jeep (red) point cloud model, after 1 iteration.

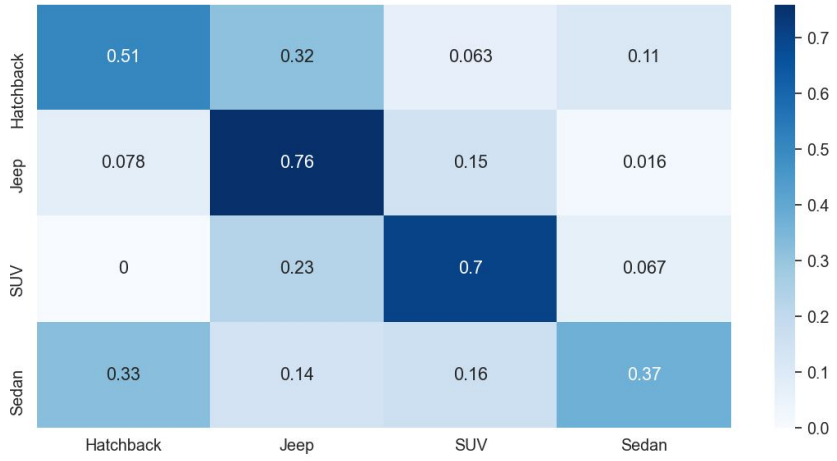

Figure 14: Confusion matrix for ICP Classification on testset ($N_{thresh} = 300$)

8.2 Point Cloud matching: Iterative Closest Point Non-Linear [2]

Iterative Closest Point Non-Linear (ICP-NL) [2] is a variant of the ICP algorithm described above. ICP-NL uses the Levenberg-Marqurdt optimization backend to solve the minimization problem as described in the previous section. This method optimizes the resultant transfomation as a quaternion, instead of as a rotation matrix, as was the case for ICP.

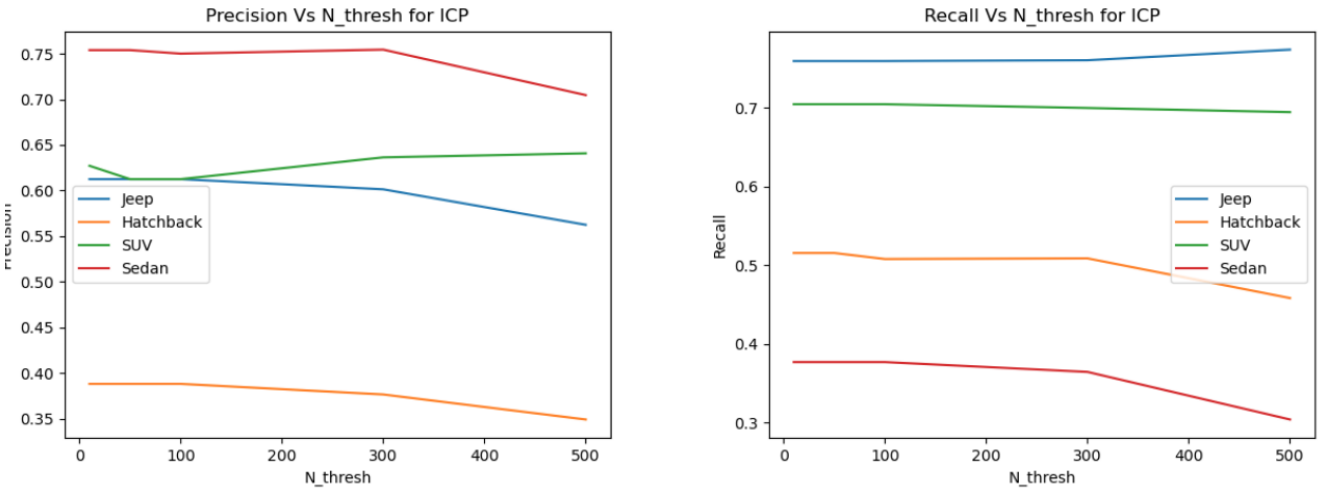

Figure 15: Precision and Recall values for all four classes over different testsets with varying N_{thresh} values for ICP Classification.

In general, ICP-NL is expected to converge faster than ICP for unstructured point clouds. In our scenario, we set up the ICP-NL classification problem exactly the same as the ICP classification problem as described in the previous section, and again used the same criterion 2 for producing predictions.

8.2.1 Results

Figure 16 shows the confusion matrix for the ICP-NL on the testset ($N_{thresh} = 300$). Comparing figure 16 to figure 14, it is clear that ICP-NL performs better in classifying all classes than vanilla ICP algorithm. ICP-NL is also better at classifying Sedans and Hatchbacks, while also further improving the results for Jeep and SUV classes. Confusion matrices for other ICP-NL Classifications on other N_{thresh} value testsets are provided in Appendix B.

Figure 17 shows the precision and recall values for ICP-NL Classification over testsets with different N_{thresh} values. Surprisingly, there is not a significant change in these values depending on N_{thresh}, which means that all the clouds that do not converge to any of the models are common in all of these testsets.

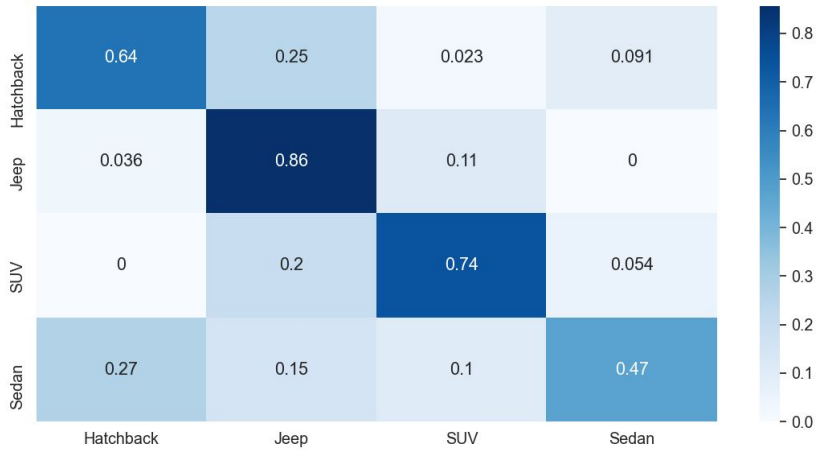

Figure 16: Confusion matrix for ICP-NL Classification on testset ($N_{thresh} = 300$)

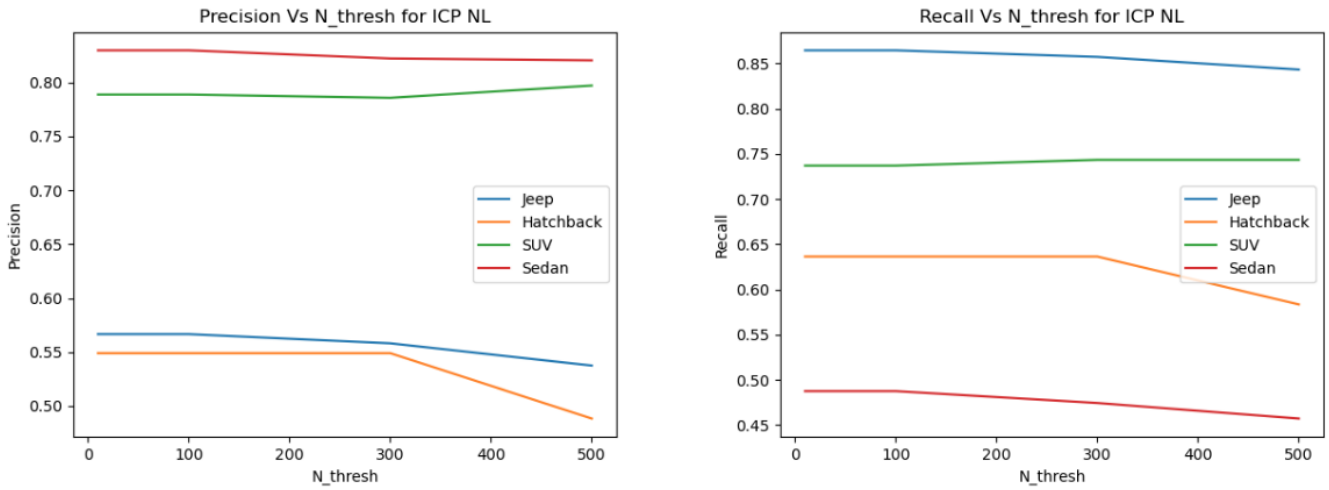

Figure 17: Precision and Recall values for all four classes over different testsets with varying N_{thresh} values for ICP-NL Classification.

8.3 Point Cloud matching: Normal Distributions Transform [3]

The NDT algorithm [3] is a registration algorithm that uses standard optimization techniques applied to statistical models of 3D points to determine the most probable registration between two point clouds. NDT, instead of trying to match all points within the clouds, sub-samples the points into voxels (3D pixels) and calculates normal distribution (mean and variance) of the points lying within each voxel. It then attepts to iteratively match these voxels between the point clouds. The result is the Rotation and translation matrices that when applied to point cloud A, results in the best alignment possible with point cloud B. NDT is generally applied to larger point clouds and is expected to work better on coarser alignments.

Since the PCL implementation of NDT algorithm is not very efficient, we were unable to iterate on this algorithm with different parameters. In our experience, NDT ran approximately 30

times slower than ICP and ICP-NL algorithms. It took 11 hours to produce results on the testset ($N_{thresh} = 100$) while running on a single-core of Intel Core i7 9700K CPU. The parameters we chose for this run were:

1. Transformation Epsilon: 0.01
    - This is the minimum transformation difference required in-between iterations for termination condition.
2. Maximum Step size: 0.1
    - NDT uses the More-Thuente line search algorithm to use the best step size below this value.
3. Maximum iterations: 35
    - The maximum number of iterations before NDT terminates. This is set so high because with the number of iterations equal to 1, we were not getting NDT to converge.
4. Resolution: 0.1
    - This is the internal Voxel grid dimension.

Similar to ICP, to classify using NDT, we iterate over all test point clouds and attempt matching them with all four models of classes (as defined in 5.1) and then use the above mentioned metrics to predict the point cloud's class.

Considering, the input as point cloud object of vehicle for which we have to predict the label and setting target as combined point cloud of each vehicle class to calculate fitness score using NDT. The class having lowest fitness score with input will be considered as label of that object.

8.3.1 Results

Figure 18 shows the confusion matrix for the classification on NDT algorithm for testset $N_{thresh} = 100$). This algorithm, by far, had the worst performance. This algorithm classifies most of the inputs as Sedans, which is why only the classification accuracy of Sedan objects is high. Due to the long running time of this algorithm, we couldn't run it multiple times to diagnose why this might have been happening. One reason could be that NDT is not suitable for smaller clouds, and that there isn't enough information in the voxel distributions to match reliably.

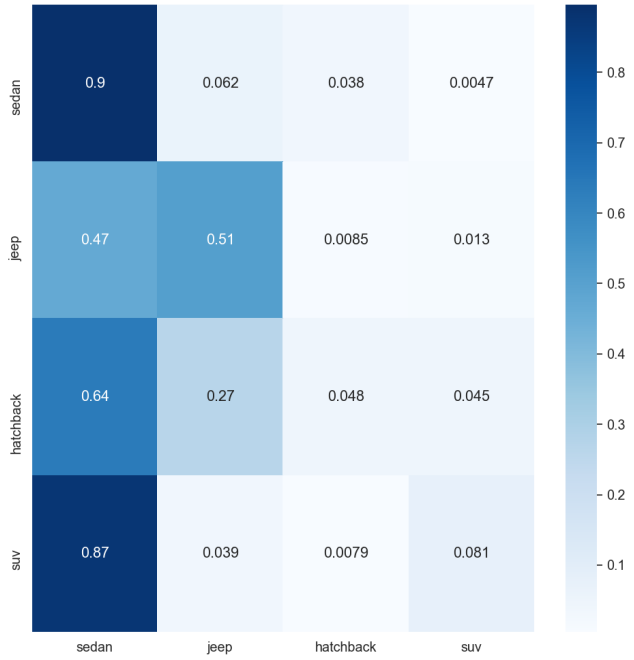

Figure 18: Confusion matrix for NDT Classification on testset ($N_{thresh} = 100$)

8.4 Unsupervised learning: K-Nearest Neighbors Classification [4]

K-Nearest Neighbors Classification is an unsupervised classification algorithm that uses instance-based learning to compute classification from a simple majority vote from the K nearest neighbors of each point. During training, each data point is assigned the class which has the most representatives within the K nearest neighbors of that point [4]. During classification, the KNN classifier returns the class that the data point most resembles to.

For us to be able to use the K-Nearest Neighbor classification algorithm, we need to represent our point clouds into feature vectors. Point clouds cannot directly be classified because each object clouds can have upwards of 20,000 dimensions (points). In order to find the correct way to convert PCDs into feature vectors, we tried visualizing some parameters. Figures 19, 20, 21 and 22 plot the means and standard deviations in the x, y, z directions of all the point clouds in the training set. As apparent from these figures, there are clear differences in the mean and standard deviation values between different classes of objects. Therefore, unsupersvised methods such as KNN should be good candidates for classification.

There are some interesting observations from the mean and standard deviation distributions:

1. The z dimension has the most consistent mean value and the least amount of variance: This is because at any given location, the LiDAR sensor can always "see" the objects completely across the z dimension. This is also due to the fact that LiDAR sensors have a vertical array of lasers.

2. The x and y dimensions have larger standard deviations: This is because the user-car is either always behind or in-front of other objects (same lane), or to the left or to the right of other objects (adjacent lanes), so observations only capture one side of the car and never the whole. This leads to higher variance.

3. The y dimension mean has a smaller range than the x dimension: This is because vehicles are almost always longer than they are wider, giving a larger range of values across x.

Appendix C plots the means and standard deviations for the PCDs in the testset. Clear similarities can be seen across classes with the training set. This gives us confidence that using these as features in the feature vectors would yield good results.

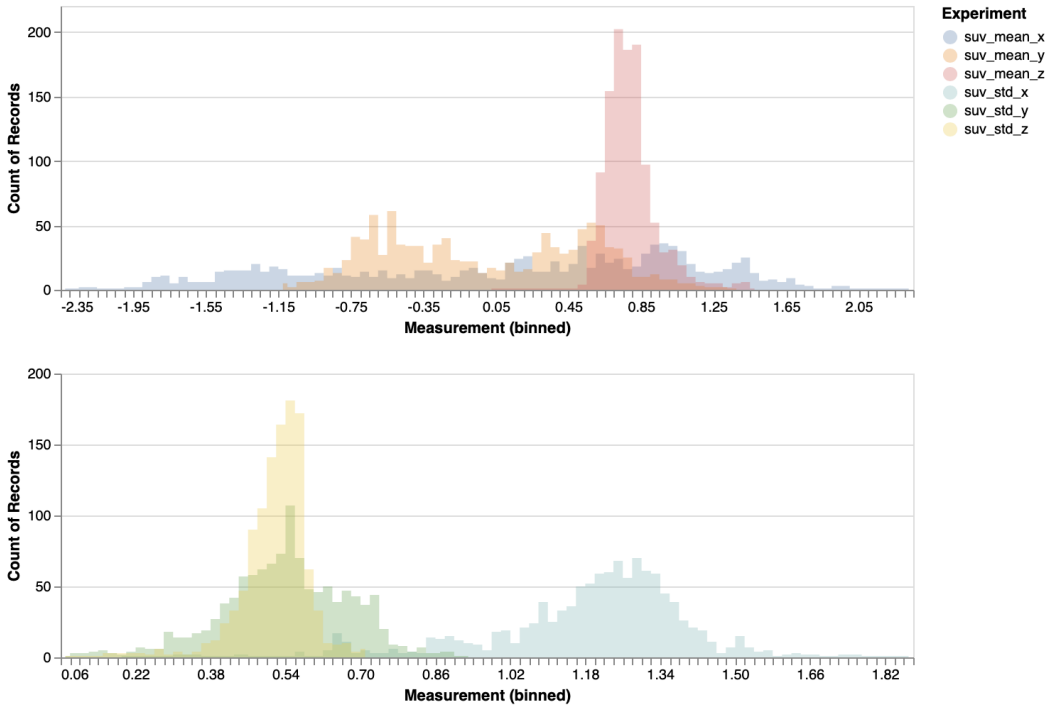

Figure 19: Visualization of the distribution of the feature vector values for all SUV point clouds in the training set.

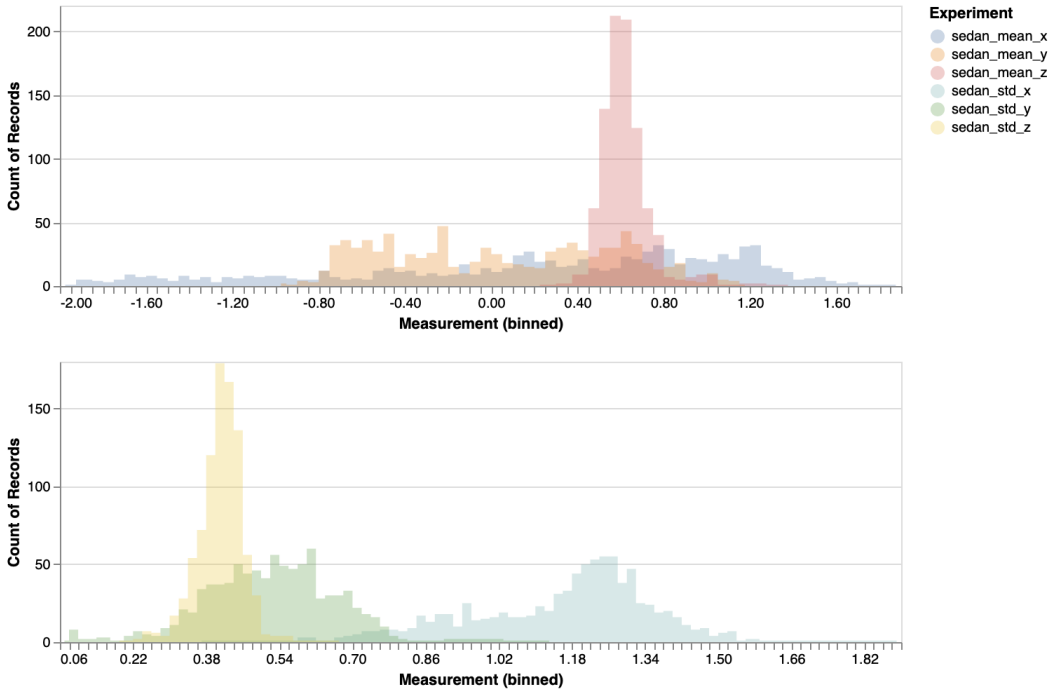

Figure 20: Visualization of the distribution of the feature vector values for all Sedan point clouds in the training set.

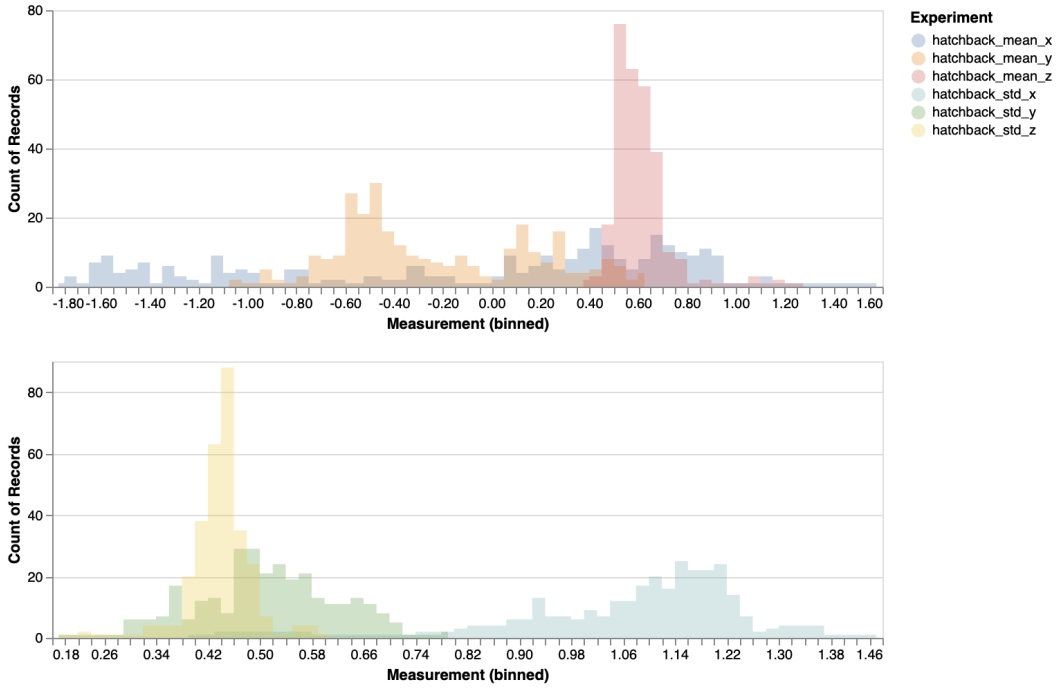

Figure 21: Visualization of the distribution of the feature vector values for all Hatchback point clouds in the training set.

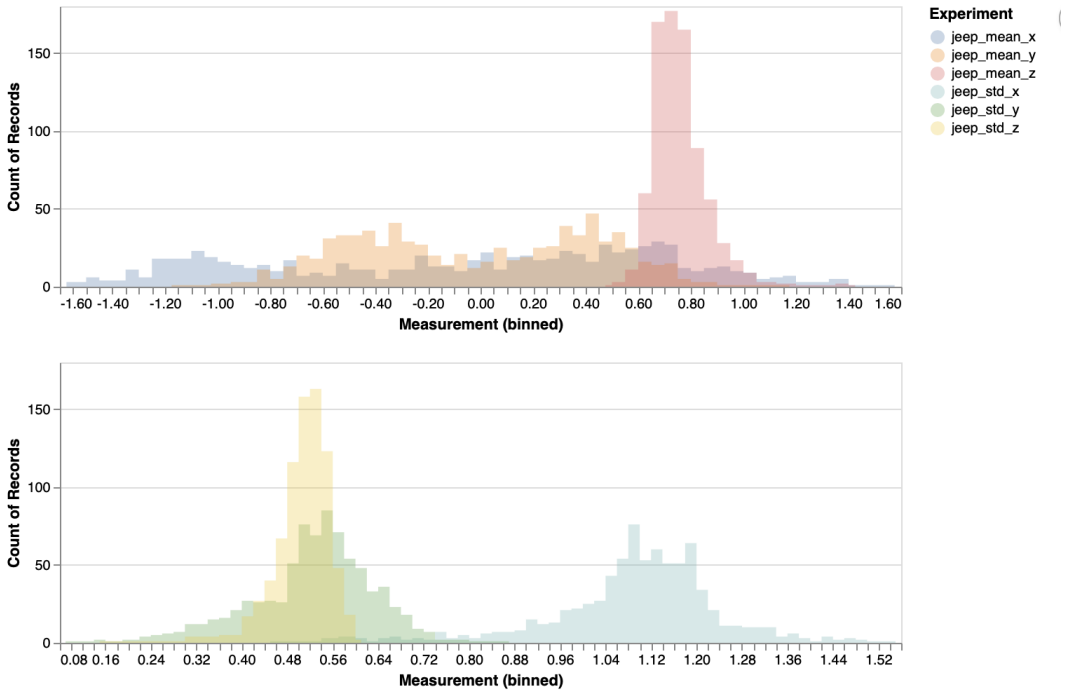

Figure 22: Visualization of the distribution of the feature vector values for all Jeep point clouds in the test training.

Therefore, we computed feature vector $\mathbf{f}_i$ for the ith point cloud pc_i as,

$$\mathbf{f}_i = [\mu_x, \mu_y, \mu_z, \sigma_x, \sigma_y, \sigma_z] \tag{1}$$

where, μ_x, μ_y, μ_z are the mean values of all points in pc_i in the x, y, z directions respectively, and $\sigma_x, \sigma_y, \sigma_z$ are the standard deviations in the x, y, z directions respectively.

We then trained the K-Nearest Neighbors classifier provided by Scikit-learn [18] with these feature vectors in the training set, and tested them on the test sets with varying N_{thresh} values. The results of the predictions are shown in Figure 23.

8.4.1 Results

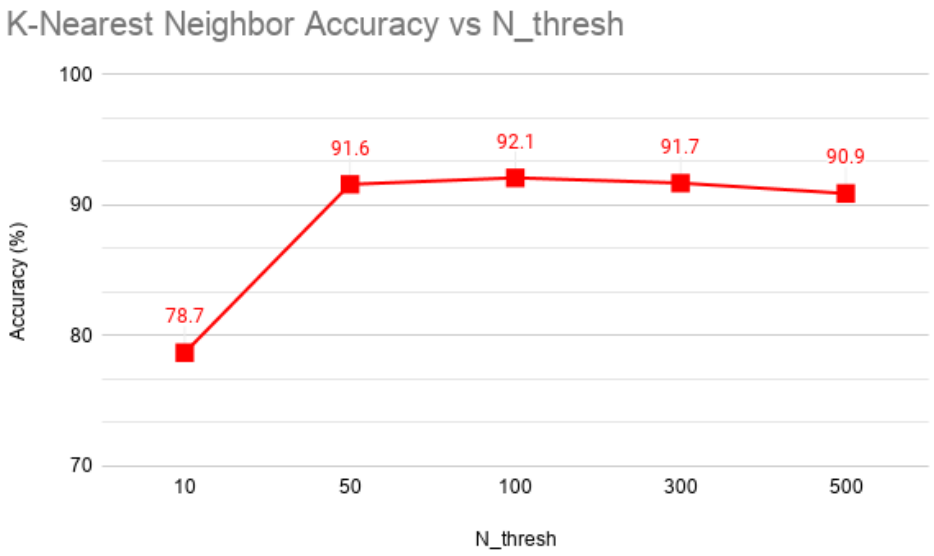

Figure 23: Shows the accuracy achieved using KNN Classification algorithm with $K = 3$ over datasets created with varying N_{thresh} value.

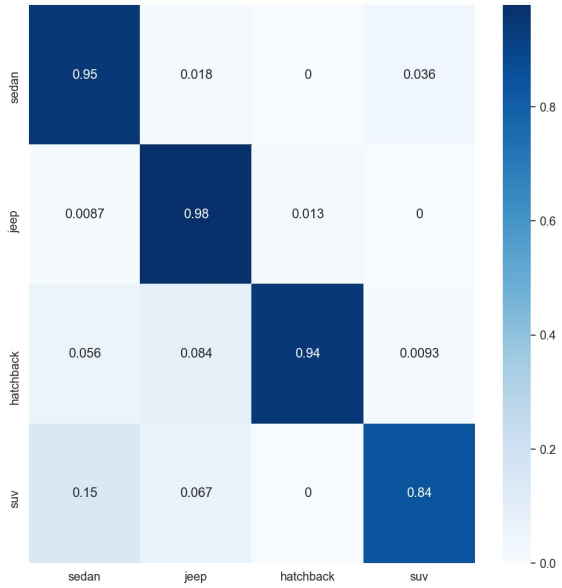

Figure 24: Shows the confusion matrix of the classification of the test set ($N_{thresh} = 100$) using KNN Classification algorithm with $K = 3$.

As visible in Figure 23, the K-Nearest Neighbor algorithm performs very well with datasets where $N_{thresh} \geq 50$. This is expected as point clouds with fewer than 50 points provide very little information about the object itself, and the mean and variance can match with other objects as well. The highest accuracy is achieved with the testset of $N_{thresh} = 100$. The accuracy actually

goes down a bit as N_{thresh} increases; this could be attributed to us ignoring some point clouds with fewer points that are in fact classified correctly.

Figure 24 plots the confusion matrix for the classification on testsets with $N_{thresh} = 100$. When compared to the confusion matrices produced by ICP for classification, KNN does a much better job at classifying objects correctly.

9 Conclusion

As per the final results, we can see that point cloud matching algorithms (ICP and ICP-NL) are good with Rigid Body Transformations and point cloud registrations, however, they are not the best when it comes to classification of point clouds. On the other hand, we observe that unsupervised classification methods such as K-Nearest Neighbors classification works very well for classifying point clouds, but cannot be used for matching or registration of point clouds with one another. Also, the performance of the K-Nearest Neighbors Classifier is dependent on the design of the feature vector.

Through our experimentation, we also realized that tuning ICP and ICP-NL algorithms for the purpose of classification is quite tricky and requires a lot of experimentation. The number of parameters to tune is quite large and getting the optimal results is not easy. KNN, on the other hand, worked well with minimal tuning.

Additionally, for point cloud matching, pre-processing the models properly plays a very critical role in the performance and speed of ICP and ICP-NL. We observed that ICP and ICP-NL work better when there are less but uniform points all over the object shape. A cluster of dense points can adversely affect the matching performance.

Finally, we attempted to use the NDT matching algorithm for classification of objects but because of the long running time of this algorithm, we were unable to iterate and achieve good results. With more time, we may be able to show that NDT can used for classification as well.

References

[1] "Documentation - Point Cloud Library (PCL)," http://pointclouds.org/documentation/tutorials/iterative_closest_point.php, (Accessed on 03/05/2020).

[2] "Point Cloud Library (PCL): pcl::IterativeClosestPointNonLinear¡ PointSource, PointTarget, Scalar ¿ Class Template Reference," https://pointcloudlibrary.github.io/documentation/classpcl_1_1_iterative_closest_point_non_linear.html#details, (Accessed on 05/12/2020).

[3] "Documentation - Point Cloud Library (PCL)," http://www.pointclouds.org/documentation/tutorials/normal_distributions_transform.php, (Accessed on 03/05/2020).

[4] "1.6. Nearest Neighbors — scikit-learn 0.23.0 documentation," https://scikit-learn.org/stable/modules/neighbors.html, (Accessed on 05/12/2020).

[5] "LGSVL Simulator — An Autonomous Vehicle Simulator," https://www.lgsvlsimulator.com/, (Accessed on 03/05/2020).

[6] "ROS.org — Powering the world's robots," https://www.ros.org/, (Accessed on 03/06/2020).

[7] "rosbag - ROS Wiki," http://wiki.ros.org/rosbag, (Accessed on 05/01/2020).

[8] "rosbridge_server - ROS Wiki," http://wiki.ros.org/rosbridge_server, (Accessed on 05/11/2020).

[9] "message_filters - ROS Wiki," http://wiki.ros.org/message_filters, (Accessed on 05/11/2020).

[10] "pcl/visualization.rst at cc7fe363c6463a0abc617b1e17e94ab4bd4169ef · PointCloudLibrary/pcl," https://github.com/PointCloudLibrary/pcl/blob/cc7fe363c6463a0abc617b1e17e94ab4bd4169ef/doc/tutorials/content/visualization.rst#pcd-viewer, (Accessed on 05/11/2020).

[11] "pcl_ros - ROS Wiki," http://wiki.ros.org/pcl_ros, (Accessed on 05/11/2020).

[12] "PCL - Point Cloud Library (PCL)," http://pointclouds.org/, (Accessed on 03/05/2020).

[13] "Point Cloud Library (PCL): Module common," https://pointcloudlibrary.github.io/documentation/group__common.html#gaff524851ffbcbefdbef2277134382906, (Accessed on 05/11/2020).

[14] "Documentation - Point Cloud Library (PCL)," http://www.pointclouds.org/documentation/tutorials/passthrough.php, (Accessed on 03/05/2020).

[15] "Removing outliers using a RadiusOutlierRemoval filter — Point Cloud Library 0.0 documentation," https://pcl-tutorials.readthedocs.io/en/master/radius_outlier_removal.html, (Accessed on 05/12/2020).

[16] "Removing outliers using a StatisticalOutlierRemoval filter — Point Cloud Library 0.0 documentation," https://pcl-tutorials.readthedocs.io/en/master/statistical_outlier.html?, (Accessed on 05/12/2020).

[17] "Documentation - Point Cloud Library (PCL)," https://www.pointclouds.org/documentation/tutorials/voxel_grid.php, (Accessed on 03/05/2020).

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

Appendices

A Confusion matrices for ICP Classifications on testsets

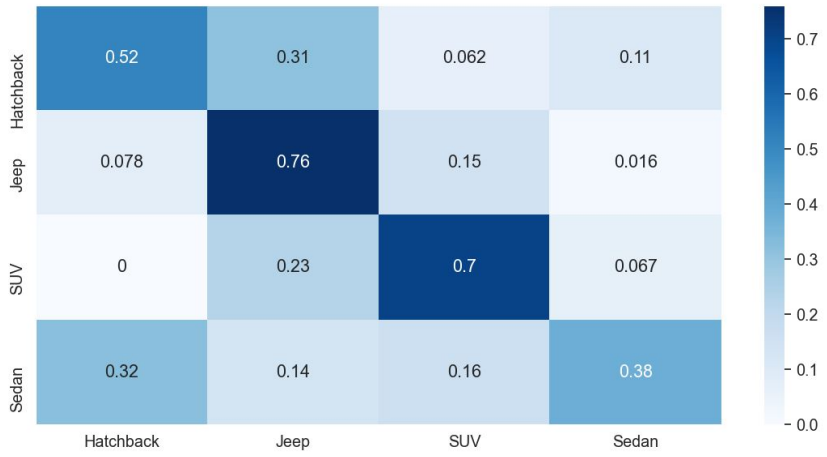

Figure 25: Confusion matrix for ICP Classification on testset ($N_{thresh} = 10$)

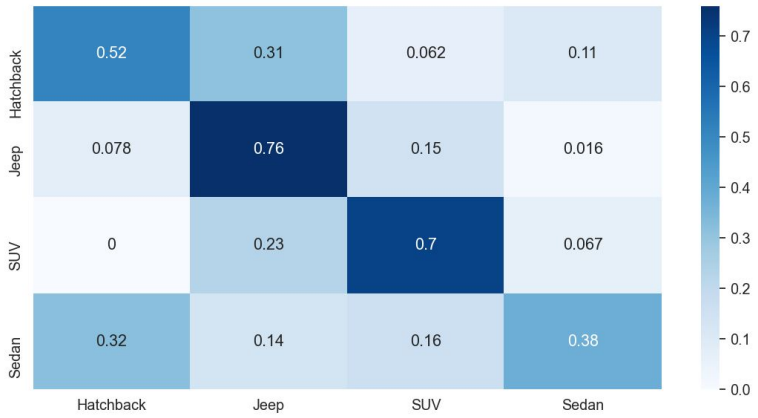

Figure 26: Confusion matrix for ICP Classification on testset ($N_{thresh} = 50$)

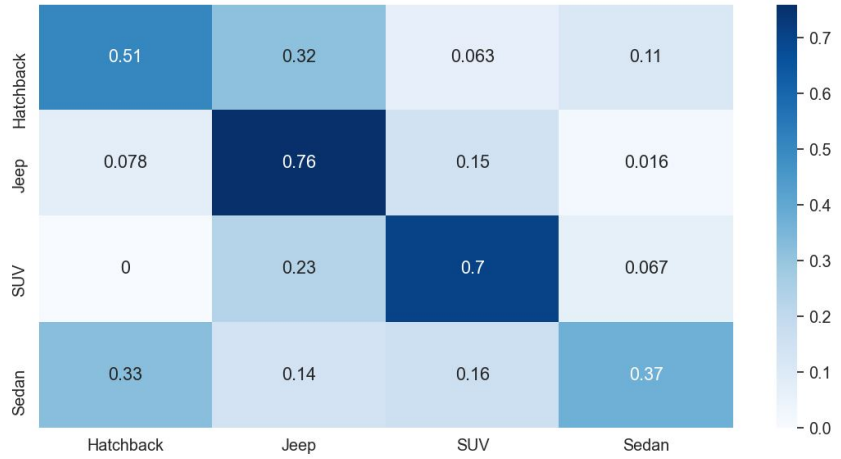

Figure 27: Confusion matrix for ICP Classification on testset ($N_{thresh} = 100$)

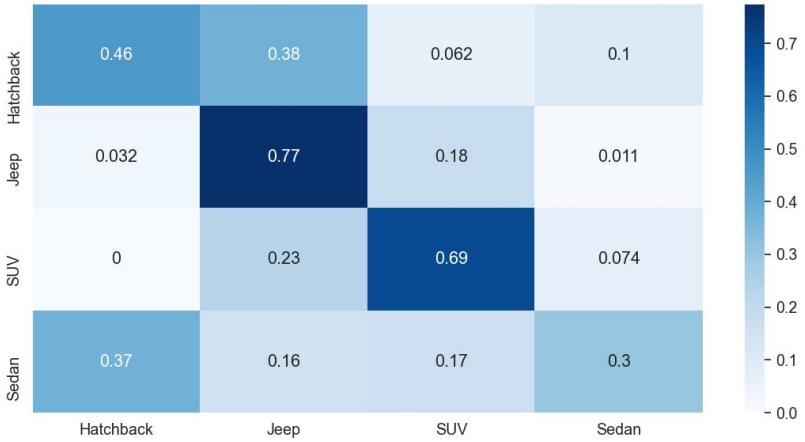

Figure 28: Confusion matrix for ICP Classification on testset ($N_{thresh} = 500$)

B Confusion matrices for ICP-NL Classifications on test-sets

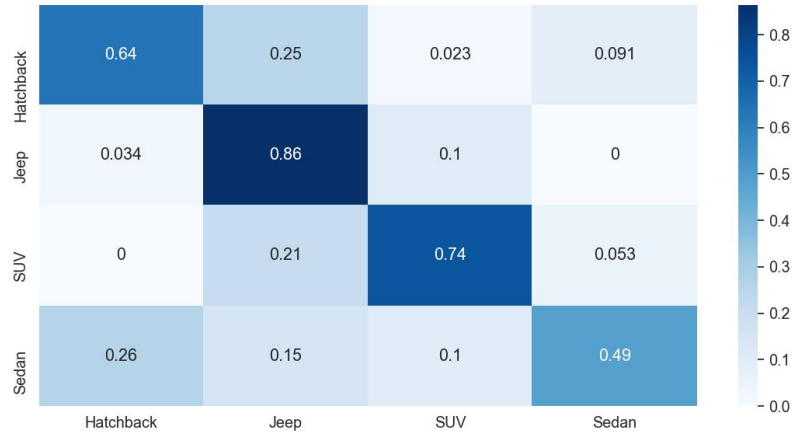

Figure 29: Confusion matrix for ICP-NL Classification on testset ($N_{thresh} = 10$)

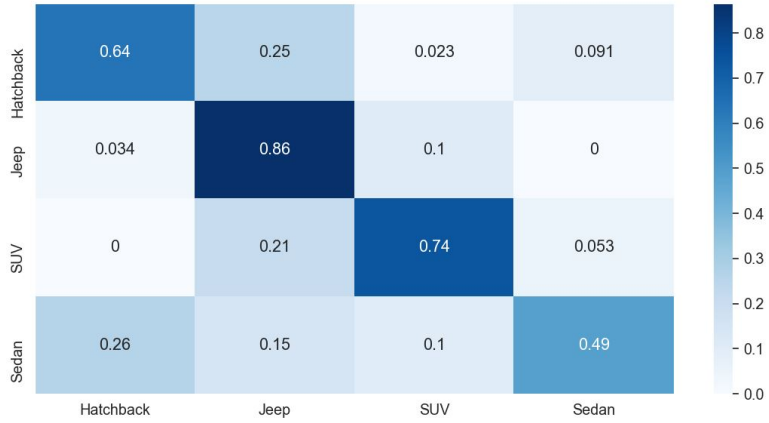

Figure 30: Confusion matrix for ICP-NL Classification on testset ($N_{thresh} = 50$)

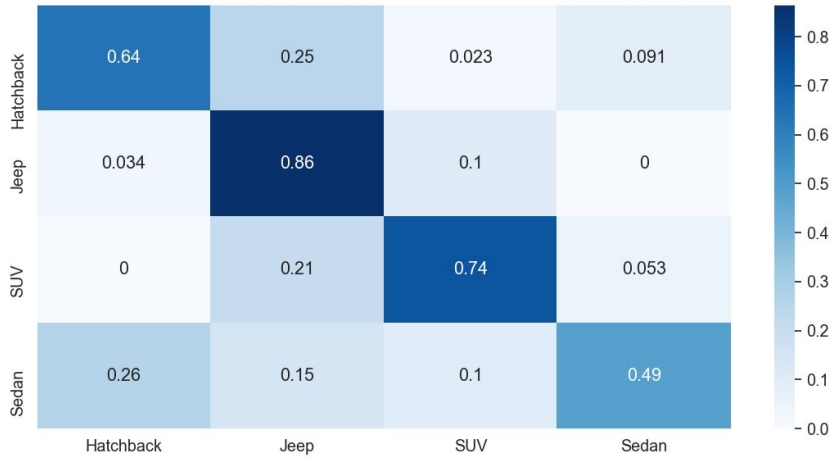

Figure 31: Confusion matrix for ICP-NL Classification on testset ($N_{thresh} = 100$)

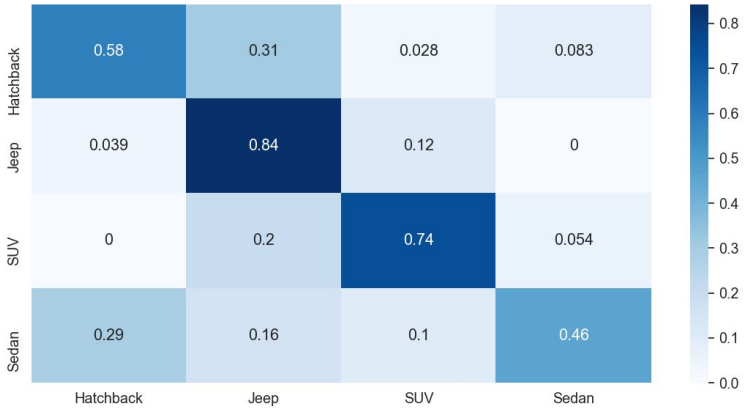

Figure 32: Confusion matrix for ICP-NL Classification on testset ($N_{thresh} = 500$)

C Visualizations of Feature vectors in the testset

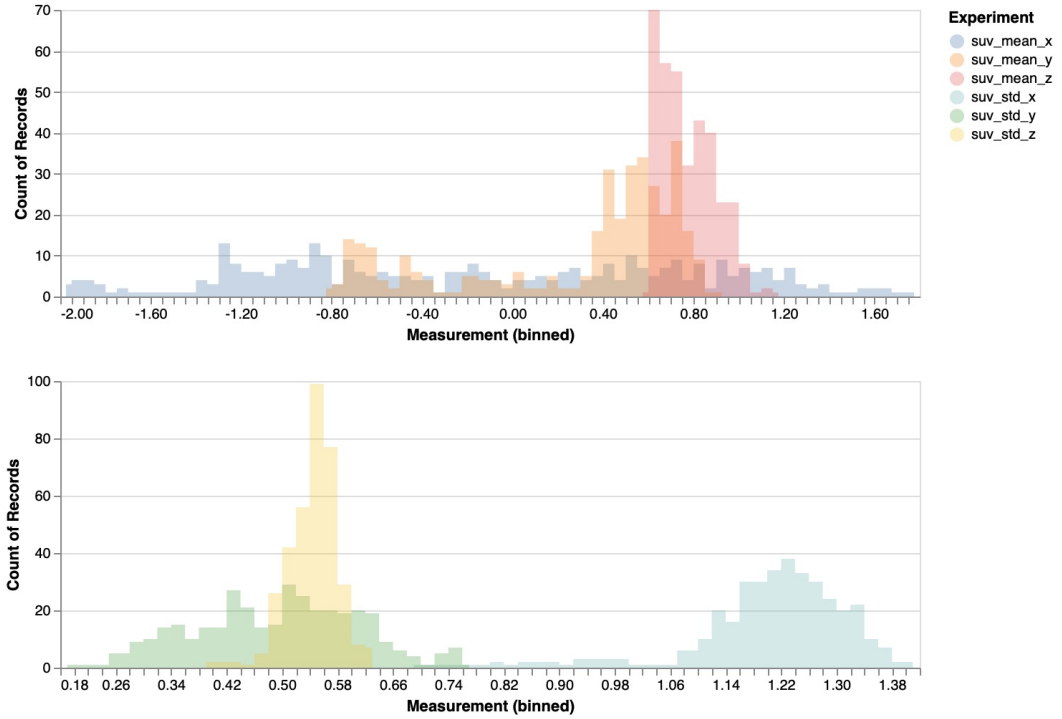

Figure 33: Visualization of the distribution of the feature vector values for all SUV point clouds in the test set $N_{thresh} = 100$.

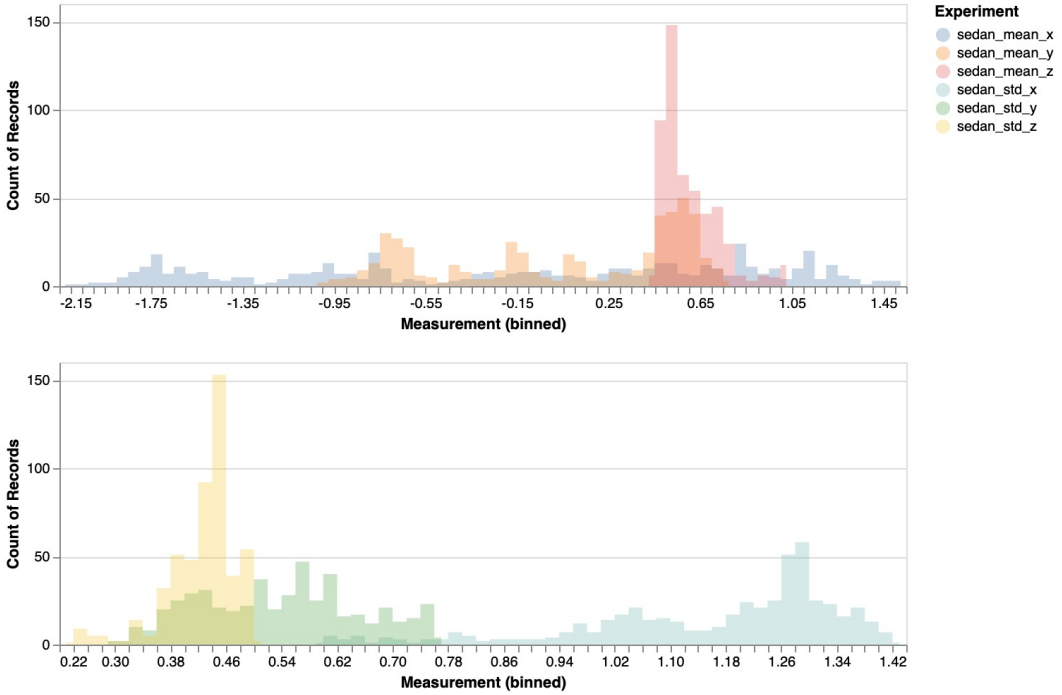

Figure 34: Visualization of the distribution of the feature vector values for all Sedan point clouds in the test set $N_{thresh} = 100$.

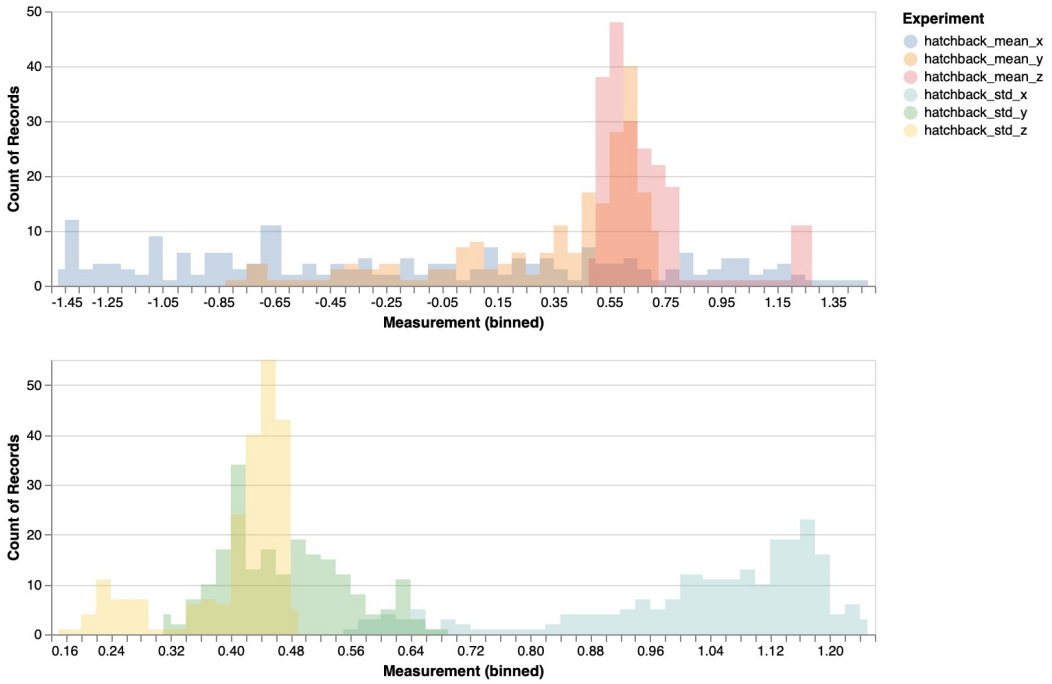

Figure 35: Visualization of the distribution of the feature vector values for all Hatchback point clouds in the test set $N_{thresh} = 100$.

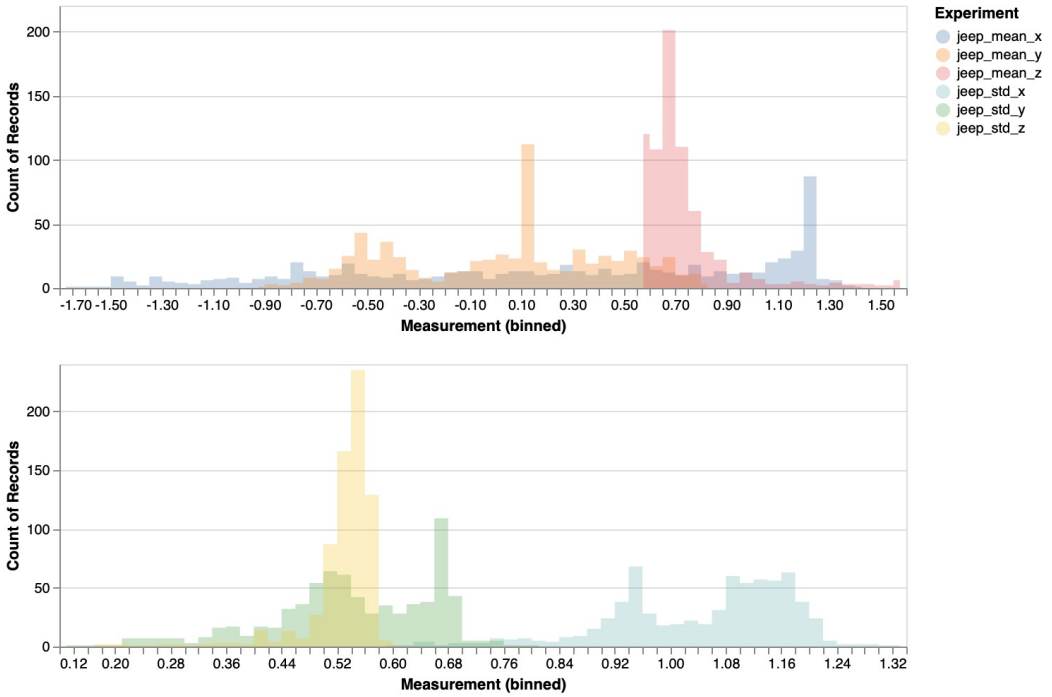

Figure 36: Visualization of the distribution of the feature vector values for all Jeep point clouds in the test set with $N_{thresh} = 100$.