3D PointCloud clustering for object classification in the context of Autonomous Driving

CMPE 255 Project Progress Report

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

May 4, 2020

1 Introduction

The objective of this project is to use labeled 3D PointCloud data collected using LiDAR sensors on Autonomous Driving platforms to develop clustering algorithms for classifying objects such as cars, pedestrians, trucks etc., as they appear in 3D PointCloud space. 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 code so far at https://github.com/deepaktalwardt/point-cloud-clustering

3 Motivation and Background

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.

4 Methodology

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 Dataset Creation

The simulation platform of our choice is the LGSVL Automotive Simulator [1]. 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) [2], which is useful for collecting, synchronizing and extracting data from the simulator. Ths simulator looks like this:

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

4.1.1 Storing into ROSbags

The first step in the dataset creation process is to collect data into rosbags [3] 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 a video. This is the best raw format to save simulated data into and is widely used in the robotics industry.

To achieve this, we connected the simulator to a ROS Master Node, launched a `rosbag record` node that would record all data being published by the simulator and store it in a `.bag` file. One such sample bag is available to be downloaded here: https://drive.google.com/open?id=1YKRs-0sgGqqfWp0IsbvrfDyRIQixvVEG.

4.1.2 Data synchronization

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 at 10Hz, we need to synchronize the point cloud and annotations time stamps 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. This process is quite tricky and important otherwise the annotations may not align properly with the point clouds.

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. 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
...
...
...
```

However, when viewed in the PCD Viewer tool, we get the following rendering:

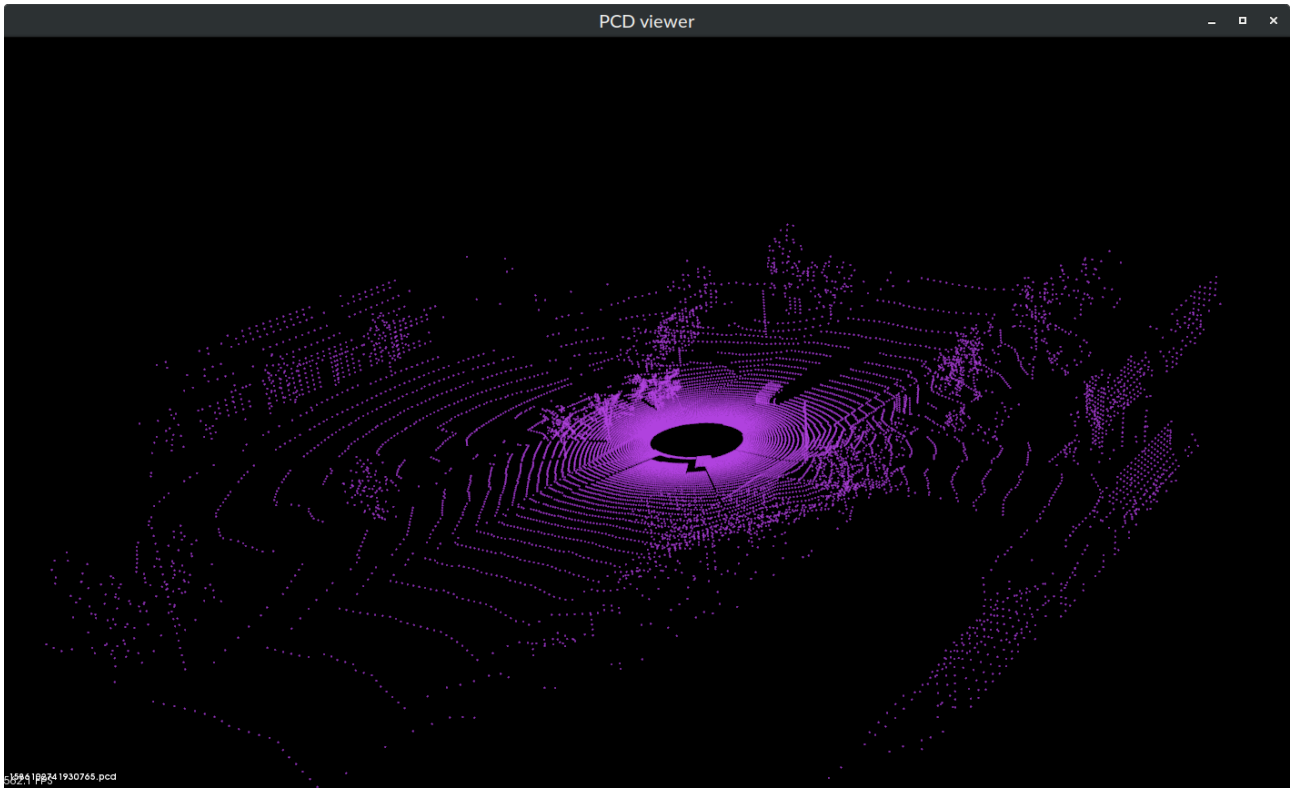

Figure 2: PCD Viewer displaying the entire point cloud

The code for saving ROS bags to PCD and JSON files is provided here https://github.com/deepaktalwardt/point-cloud-clustering/blob/master/ros_ws/src/dataset_generation/scripts/bag2files.py

4.2 Data Preprocessing

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

4.2.1 Extraction of objects

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) [4] 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. This crop box is then extracted from the point cloud and saved as a `.pcd` file. We repeat this process for all detections in all point clouds. This gives us all objects as individual point clouds. Example of extracted car point cloud:

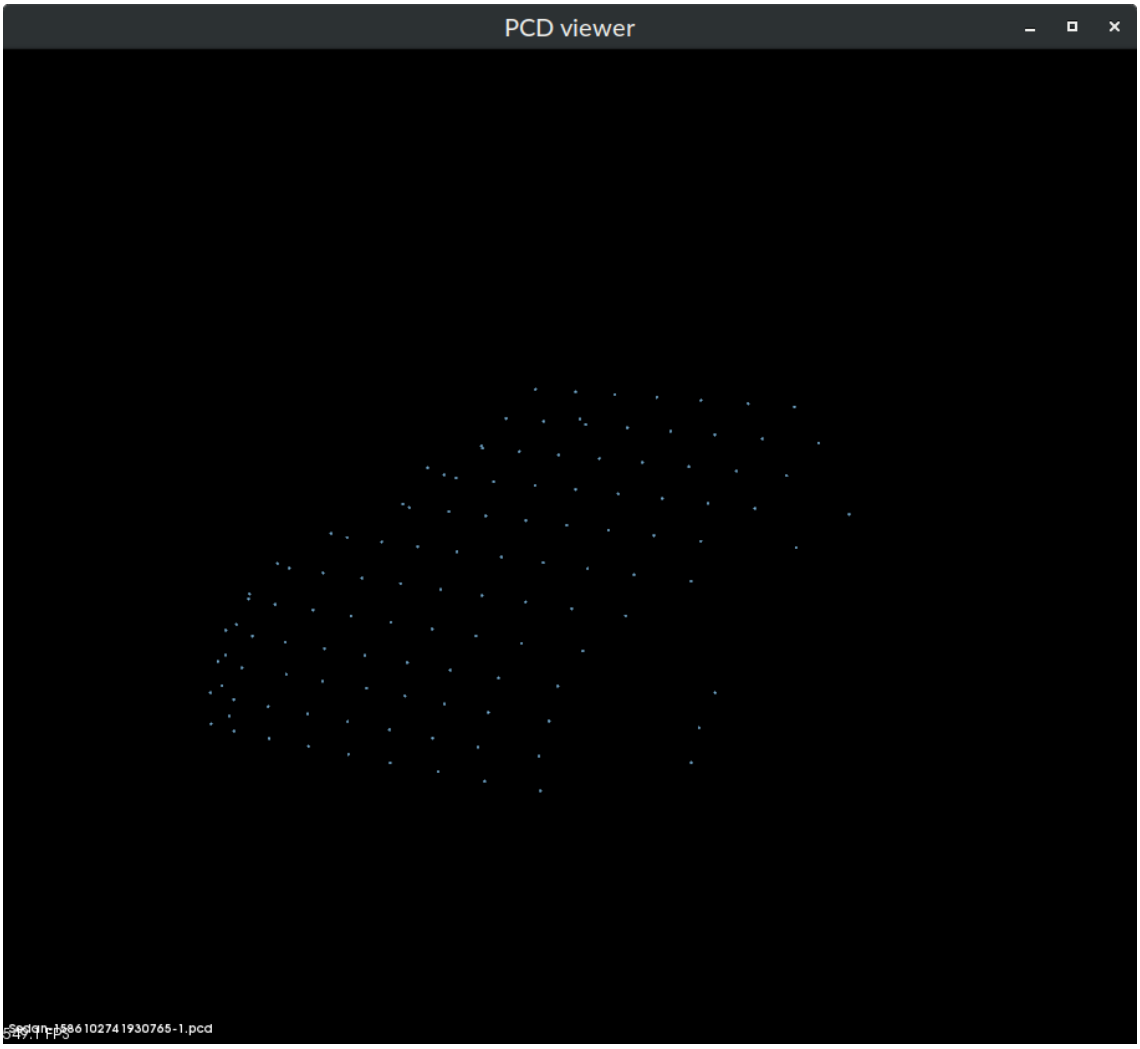

Figure 3: PCD Viewer with a single "Sedan" object

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

1. https://github.com/deepaktalwardt/point-cloud-clustering/blob/master/ros_ws/src/dataset_generation/src/extract_point_cloud_objects_node.cpp
2. https://github.com/deepaktalwardt/point-cloud-clustering/blob/master/ros_ws/src/dataset_generation/include/dataset_generation/extract_point_cloud_objects.h

4.2.2 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 are fixing currently. Here is what the visualization looks like currently.

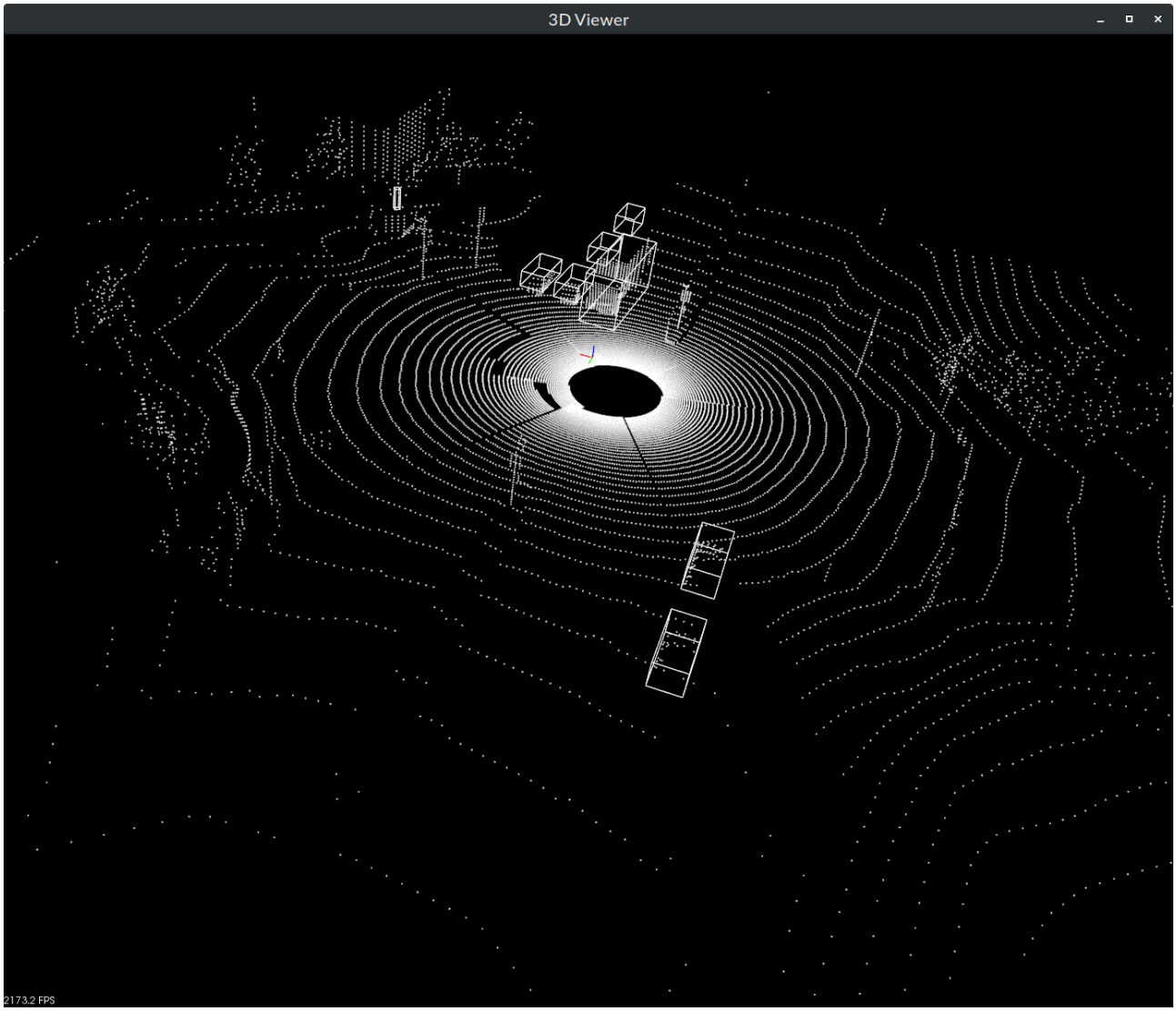

Figure 4: Visualization tool that draws boxes around the objects

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

4.2.3 Inverse transformation and concatenation

After all the objects have been extracted, we perform a pruning step where we ignore all clouds with fewer than 100 points. This is because these objects don't provide enough information and recreating an object from fewer points is not possible.

As these point clouds stand right now, they cannot be concatenated together to form the model as these points are located with reference to their original frames of references that they were initially extracted from. These points will need to be translated to 0, 0, 0 and rotated to the same orientation so that they can be concatenated. This is done using the transformation functionality provided by PCL and using the inverse of the locations and orientations of the detections. After doing this, we can concatenate these points to create models.

5 Next Steps

5.1 Sub-sampling using Voxel Filtering

Voxel filtering is a technique that is used to sub-sample points in 3D Voxel (3D pixels) space. This will be needed because the models we create from the point clouds are likely to be very dense, and point cloud matching techniques work well only when the number of points in the point clouds being matched are in the same order of magnitude. This will also be done using PCL library in C++.

5.2 Point Cloud Matching using ICP

Iterative Closest Point (ICP) [5] is an iterative point cloud matching algorithm that finds the transformation (Rotation and translation) between two point clouds. That is, it returns the Rotation and translation matrices that when applied to point cloud A, results in the best alignment possible with point cloud B. We will be using this algorithm to determine how similar two point clouds are.

5.3 Point Cloud Matching using NDT

Normal Distribution Transform (NDT) [6] is a point cloud matching algorithm that instead of trying to match all points, sub-samples the points into voxels (3D pixels) and calculates normal distribution (mean and variance) of the points lying within each voxel. It then matches 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. We will be using this algorithm to determine how similar two point clouds are. NDT is expected to run faster than ICP.

5.4 Classification

To be able to classify an unseen point cloud object, we will attempt matching it with all the models of the sub-sampled point clouds that we have. We will fix the number of maximum iterations allowed for matching and then output the resulting mismatch error. This error measurement is a good metric for determining the quality of the matching. We will then report the class of the object as the class that it receives the lowest matching error with.

6 Responses to Prof. Rojas' questions

6.1 How large are these point clouds? Can the system(s) that you have access to handle to load? (CPU, Memory, etc)

The collected point clouds contain anywhere from 15000 to 50000 points, and range from 2-3 MB to 10 MB in size. Since we will not be having a large number of these point clouds in memory at any given time, it should be manageable. The system we have access to is powered by 8-core Intel Core i7 and has 32 GB of RAM.

6.2 Does this data have video? Would it be easier to cross-reference that to help?

Yes, the simulator we are collecting data from also has a simulated camera that streams images along side the LiDAR point clouds. However, the main purpose of this project is to observe and analyze how objects such as cars, jeeps, trucks etc., "look" like from the view of a LiDAR point cloud, so the image data won't be useful here. We will still be using image data for debugging purposes though.

6.3 Another thing you could do is focus on a binary classification (car or not car). Could simply things.

Yes, if we are unable to achieve good accuracy with multi-class classification, we will reduce the problem to a binary classification.

6.4 What sort of cleaning/filtering do you think you will need to do to analyze the data?

Data collection and preprocessing are huge chunk of our project and are multi-stage processes that are explained in Sections 4.1 and 4.2.

References

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

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

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

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

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

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