MASTER'S THESIS
Centre for Mathematical Sciences

LU-CS-EX: 2019-XX

LiDAR Pedestrian Detector and Semi-Automatic Annotation Tool for Labeling of 3D Data

Erik Andersson, Roy Andersson

LiDAR Pedestrian Detector and Semi-Automatic Annotation Tool for Labeling of 3D Data

Erik Andersson
erik.andersson.811@student.lu.se

Roy Andersson
roy.andersson.269@student.lu.se

August 27, 2019

Master's thesis work carried out at
Centre for Mathematical Sciences, Lund University.

Supervisor: Mikael Nilsson, micken@maths.lth.se

Examiner: Mattias Ohlsson, mattias.ohlsson@thep.lu.se

Abstract

The goal of this Master's Thesis is to successfully detect and classify humans in a LiDAR data stream. The focus of the Thesis is on the detection and classification, not on the LiDAR technology.

To classify humans machine learning was used and to train the machine learning model we collected our own data and annotated it. A custom software was made for speeding up the annotation process.

The process for detecting humans in a scene was to first sweep the scene with a fixed size box which contain a point cloud. These point clouds were then split up by a clustering algorithm. Finally features were extracted from the clusters and classified using a classification algorithm.

The algorithm of choice for prediction became the Random Forest classifier which successfully classified unobstructed humans in different environments but occasionally gave false positives.

Contents

1 Introduction **5**

2 Sensor **7**

3 Data Gathering **11**
3.1 Overview of the Annotation Process 11
3.2 Plane Removal 12
3.3 Clustering 13
3.4 Optical Flow 15
3.5 Sweeping Annotator 20

4 Classification **21**
4.1 Training process 21
4.1.1 Overview of the Training Process 21
4.1.2 Voxel Grid 21
4.1.3 Features 23
4.1.4 The Feature Vector 24
4.1.5 Classifiers 24
4.1.6 Random Forest 24
4.1.7 Support Vector Machine 25
4.1.8 K-Nearest Neighbors 28
4.2 Classification Process 31
4.2.1 Overview of the Classification Process 31
4.2.2 Identifying Interesting Clusters 31
4.2.3 Prediction and Non-Maximum Suppression 31
4.3 Evaluation of classifier 34

5 Results **37**
5.1 The Sensor 37
5.2 Plane Removal 39

5.3 Annotator 44
5.4 Classifiers 48
5.4.1 The classifiers in general 48
5.4.2 Random Forest 48
5.4.3 Support Vector Machine 54
5.4.4 K-Nearest Neighbors 58
5.5 Features 61

6 Discussion **65**
6.1 General Discussion 65
6.2 Classifiers 65
6.3 Features 66

7 Future Work **67**
7.1 Annotation Tool 67
7.2 Classifier 68

8 List of Changes **71**

Chapter 1

Introduction

LiDAR has been a fast growing technology recent years, mainly because of its use in autonomous vehicles and for mapping terrains. The aim of this project is to create a basic pedestrian detector using LiDAR and machine learning and to create a labeling tool to create a quick and easy data gathering process.

The project has three main parts; The annotation process where data is collected and labeled, the training process where the machine learning models are created and trained and lastly the classification process where pedestrians are identified and marked in a point cloud.

The annotation process is a semi-automatic annotation tool created for labeling of objects and is especially effective for moving objects. In this project it is mainly used for annotation of humans as it can quickly annotate large amounts of humans compared to a manual annotator. The annotator (concept is based on [1]) has two major parts, the isolation of clusters (based on [2]) and the tracking of a cluster between frames.

In the training process three different classifier algorithms are implemented and compared, these are Support Vector Machines, Random Forests and K-nearest neighbors classifiers. They are evaluated by comparing the accuracy and time required for a prediction. The features used are a combination of voxels and other hand-crafted features.

The detection of pedestrians is based on works such as [3], [4], [5] and [6]. Which have used various ways of detecting pedestrians from a LiDAR point cloud.

The next chapter (chapter 2)briefly describes the LiDAR technology as well as the sensor used. Chapter 3 describes the semi-automatic annotation tool and includes the algorithms used. Chapter 4 describes how the detection of a pedestrian is performed, this includes the feature extraction, classification and how clusters were extracted from the LiDAR data.

Chapter 2

Sensor

A LiDAR works by emitting a pulsed laser and by measuring the time of return and change in wavelength of the reflection. Using the time between the emission of the the pulse and the return it is possible to calculate the distance to the object reflecting the pulse. By having multiple sensors and by slightly moving them between measurements it is possible to create a three dimensional point cloud of a scene as can be seen in Figure 2.2. Some of the more common applications for LiDAR are mapping of landscapes and object detection on autonomous cars.

The sensor used in this project has a range of 200m with 0.2°of spatial resolution. Unlike most other LiDARs on the market the sensor is not rotating and has a field of view of 60x24°. It operates at a wavelength of 905nm. It has a 10Hz sample rate and is sampling 540 000 points per second.

After converting the sampled data to csv files, every file contains about 22 000 points. The LiDAR is accompanied with a Software Development Kit (SDK) which was mainly used as a tool to capture data and to convert the data to csv files.

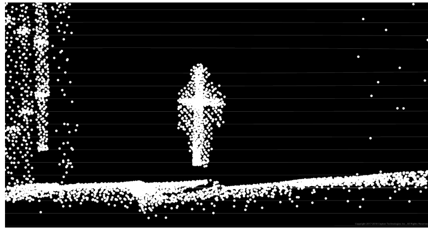
The sensor consists of a grid of 24 (3x8) channels which are combined to create the point cloud. There is an overlap of up to 10% between each channel (this can be seen in Figure 2.2 and in 2.1a as the cross on the human cluster), this creates areas which have higher concentrations of points resulting in an area with higher resolution.

The output values from the LiDAR can be seen in Table 2.1, the values that are relevant and used in this project are the x, y and z-coordinates and the intensity.

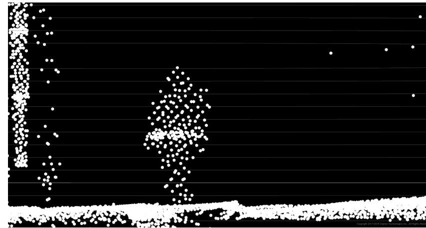
A part of the project has been to test the LiDAR to see what restrictions there are on the sensor. Some general tests such as how it reacts to different types of weather, how different surfaces such as glass and mirrors are perceived and how it works at different ranges as well as any other general restrictions with the technology.

# timestamp_usec	image_x	image_z	distance	x	y	z	azimuth	elevation	intensity	return_strongest	return_farthest	valid	saturated
1560250463085661	0.469	-0.21	15.972	-6.661	14.208	2.98	0.438	0.188	0.33	1	1	1	0
1560250463085664	0.294	-0.202	15.262	-4.224	14.375	2.903	0.286	0.191	0.64	1	1	1	0
1560250463085667	0.134	-0.196	14.818	-1.935	14.416	2.832	0.133	0.192	0.43	1	1	1	0

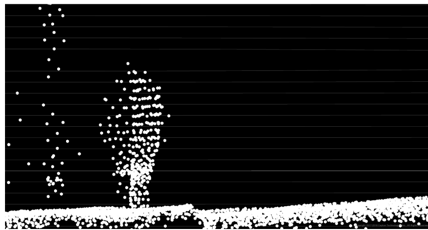
Table 2.1: Extract from a csv file created using the SDK



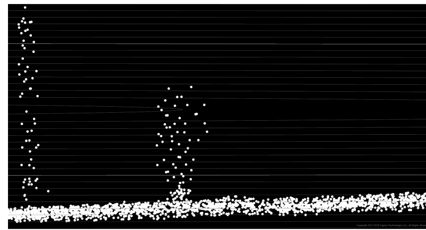
(a) Human cluster 5m from sensor. The person is so close to the sensor that the lower part of the cluster is missing



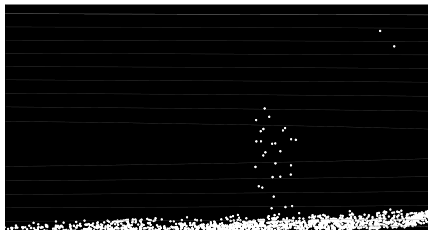
(b) Human cluster 10m from sensor, there is a noticeable difference in point density from 5m.



(c) Human cluster 15m from sensor. Half of the cluster is in the overlap between channels where the density of points is higher.



(d) Human cluster 20m from sensor. At this distance it is becoming more difficult to decide whether the cluster is a human.



(e) Human cluster at 25m. It is now very difficult to decide whether the cluster is a human, the whole cluster contains about 20 points.

Figure 2.1: Human cluster at different distances from the sensor.

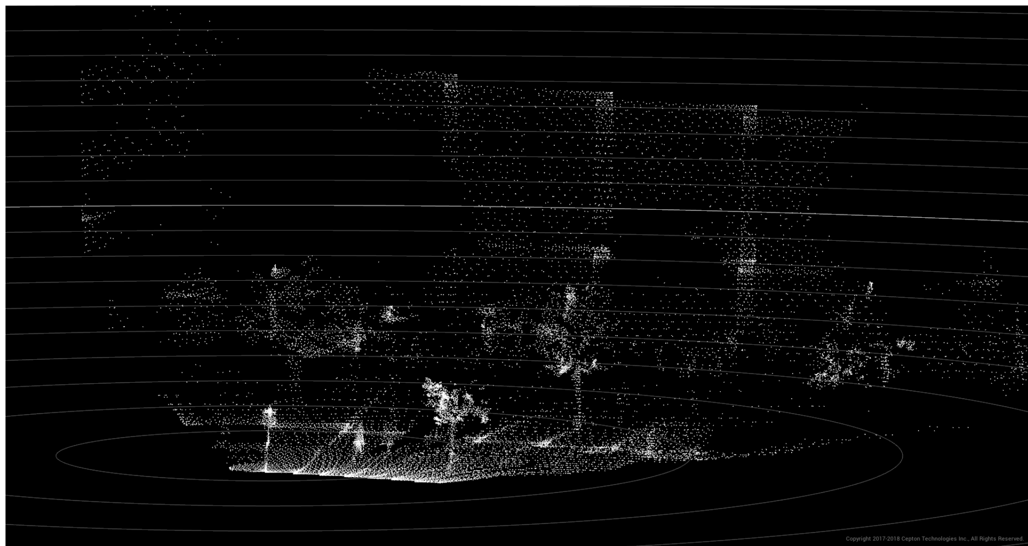


Figure 2.2: An image captured with the LiDAR, it is a scene containing two pedestrians, some trees and some buildings in the background. The overlap between channels can be seen on the buildings as higher point density areas.

Chapter 3

Data Gathering

3.1 Overview of the Annotation Process

The first part of the project is to annotate (or label) data to be used as training data and testing data. Data annotation can be one of the most time consuming processes in machine learning, especially if done manually. Since the LiDAR that is used in this project is relatively new there is not any pre-labeled data to find and because of the importance that the labeled data used for training comes from the same sensor as the test data there was no option to use data from other sensors such as the commonly used LiDAR datasets from KITTI Vision Benchmark Suite. However, during the process of this project, Herzog and Dietmayer [7] proposed a method to train a lower resolution model using data from a higher resolution sensor. To significantly decrease the time and workload of data labeling, two semi-manual processes were implemented. One of the annotators is used to label human clusters while the other one is used for object clusters.

The raw data received from the LiDAR is converted to csv files (see Table 2.1 for an example) by using a script included in the SDK. Each csv file is considered a frame. From the csv files the x,y,z-coordinates and intensity of each point were extracted.

The next step is to create clusters of points within a certain distance of each other, this was done by using Density-Based Spatial Clustering of Applications with Noise (DBSCAN) (see 3.3) on the x and y coordinates of the points (a cluster of two dimensions instead of three). There is however a problem when using DBSCAN on the unprocessed data; since only the x and y coordinates are used, a ground or ceiling plane would connect all clusters. Therefore the major planes of the point cloud were identified in one frame and then removed from all other frames. The process of plane removal followed by clustering was inspired from [2].

Once the planes are removed the labeling process starts, here the person annotating has to visually decide what kind of object each cluster is, most of the data used here has been acquired in a controlled setting (large open area with no object close to the human to be annotated) to make this step as easy as possible. A bounding box is created around each cluster,

it is important that the cluster (bounding box) includes all points of an object. Optical flow is used to avoid having to manually label every cluster in every frame, optical flow is able to keep track and follow individual clusters between frames. The job of the human annotator is to, at every frame, visually inspect the clusters and make sure that all points are within the bounding boxes and part of the cluster. The idea of tracking humans and automated labeling them was sparked when reading [1], the method is slightly different but the idea is the same.

The human annotator is complemented with an object annotator. This annotator is used on scenes that only contain object (i.e. no humans) and classifies all clusters within a box that is swept through the scene as an object.

The human annotation process can be summarized as:
1. Plane removal.
2. Clustering using the x and y values of the point cloud.
3. Labeling of clusters.
4. Tracking of the clusters between frames using optical flow -> 3.

The object annotation process can be summarized as:
1. Create a box that is swept through the scene.
2. Clustering of points inside of the box.
3. Labeling of clusters larger than 20 points.

3.2 Plane Removal

Plane removal was done by finding planes using Random Sample Consensus (RANSAC). RANSAC can find planes reasonably fast depending on which parameters are set and the size of the data. However to always find the correct plane and automatically remove it was proven difficult for this project, therefore a GUI was made for manually observe which plane was chosen before removing it. The RANSAC algorithm has five input parameters that can be set:

Parameter	Description	Default value
data	original data points	all points
num	min points for finding a model	3
iter	max amount of iterations before terminating	500
distance	max distance between the plane and a point	0.001
inlierratio	min amount of inliers needed for a good model	-

In the case of this project **num** was always 3 since only plane fitting was of interest and therefor removed as a parameter. **inlierratio** was also removed, the best plane was always returned, no matter if it was bad or good. The following implementation of RANSAC which was used in this project is based on the algorithm presented in [8]. RANSAC can be summarized in the following steps:

1. Pick a random sample of **num** size from the **data**
2. Fit a model to that sample, in this project 3 points were used to fit a plane.

3. Count how many points are within **distance** of the plane. Those points are called inliers
4. Repeat step 1-3 **iter** times.
5. Return the plane with the most inliers.

A more modern approach to RANSAC can be found in [9] but it was never used for this project. RANSAC is good because it is robust against outliers which there are many of in a normal scene. It can also run fast when decent parameters are set. However it is hard to make the process fully automatically because the parameters you set are typically depending on the scene and if there is no prior knowledge of the scene it will be impossible to set perfect parameters. If some knowledge of the scene is known one can set which regions to find a plane, e.g. only fit a plane to the points at the bottom of the scene to find the floor or at the top to find the ceiling. This works very well and therefor the GUI supports setting which regions to find planes in and get a visual overview of the scene. However it was observed that in office environment there were often a lot of objects hanging from the ceiling which obstructed the scene such as lamps and wires. These were unwanted objects which were removed by setting a threshold at a certain height, e.g. 2 meters, and everything above that threshold got removed. Since that obviously also varies from scene to scene it is up to the operator of the GUI to chose which threshold is suitable. To simplify for the operator of the program a button for finding a floor plane was added. It tries to automatically find the floor plane which is the plane that always needs to be removed no matter the scene. This is done by performing RANSAC on all points that is between 0% and 10% height from the bottom of the scene. Then the normal of the plane is checked to see that the angle seems to be correct. Once a floor plane is found, the plane is risen up by a few centimeters and then everything below the plane is removed. This is done to remove as much noise as possible. Of course one could, just like with the ceiling, set a hard threshold at a certain height and remove everything below, the problem with that solution is that the LiDAR, which normally is placed quite high up and at an angle, makes all floor planes look like they are at an angle. This could end up with parts of or even entire clusters being removed.

After the operator is satisfied with the ceiling threshold and which planes to remove, the values for the planes and threshold is saved and all points above the threshold or within the planes are removed for every frame in the scene. This works well as long as the LiDAR is placed stationary at one position and not moved around. If the LiDAR is not stationary the method would not work since the scene would change from frame to frame.

3.3 Clustering

This entire section is based on the original article from [10], for a more thorough explanation the reader is encouraged to read the original article.

Density-based spatial clustering of applications with noise (DBSCAN) is a clustering algorithm that only require two parameters:
1. ϵ which is the maximum distance between two points required to consider them as belonging to the same cluster.
2. Minimum amount of points which is the minimum amount of points required inside a cluster to consider it as a cluster.

Optionally one could also set a parameter representing the distance function, that is the

function that calculates the distance between two points. Normally, at least for 2D and 3D space, the distance function is the euclidean distance function.

The main advantage, which is why it was chosen for this project, is that there is no requirement of specifying how many clusters DBSCAN is supposed to find. It will find an arbitrarily amount of clusters as long as they meet the requirements of ϵ and minimum amount of points. The disadvantage is that the user has to manually set both ϵ and minimum amount of points which requires the user to have some prior knowledge and understanding of the dataset. It can present some problems when one wants to automatize a system without any need of any user to go in and monitor or telling the system what is correct. The run time of DBSCAN in average is $O(n \log n)$ with a worst case run time of $O(n^2)$ The algorithm can be summarized in 5 steps:
1. Chose a random point P in the dataset $Points$
2. Loop over all points in $Points$ and add every point that is within ϵ range to $neighbours$
3. If size of $neighbours$ is less than $minpts$ label it as $Noise$ otherwise label it as a $corePoint$
4. Cluster together all $CorePoints$ that are within ϵ vicinity of each other.
5. Repeat until all points are labeled as either a cluster or noise In Figure 3.1 there is a visual representation of the DBSCAN algorithm in 2D. Note however that DBSCAN works in any dimension as long as the distance function is set properly for that dimension. Following is the pseudocode for DBSCAN.

Algorithm 3.1: DBSCAN.

```
1  DBSCAN(points, eps, minPts)
2     C = 0
3     for each point P in points
4       if label(P) != undefined then continue
5       Neighbours N = FindNeighbours(points, P, eps)
6       if size(N) < minPts then
7          label(P) = Noise
8          continue
9
10      C = C + 1
11      label(P) = C
12      set S = N \ {P}
13      for each point Q in S
14         if label(Q) = Noise then label(Q) = C
15         if label(Q) != undefined then continue
16         label(Q) = C
17         Neighbours N = FindNeighbours(points, P, eps)
18         if size(N) ≥ minPts then
19            S.add(N)
```

Algorithm 3.2: FindNeighbours.

```
1  FindNeighbours(points, Q, eps)
2     Neighbours=empty list
3     for each point P in points
4       if distanceFunction(Q,P) ≤ eps
5         Neighbours.add(P)
6
7     return Neighbours
```

Where distanceFunction on line 4 normally is the the distance function that calculates the euclidean distance between point P and point Q.

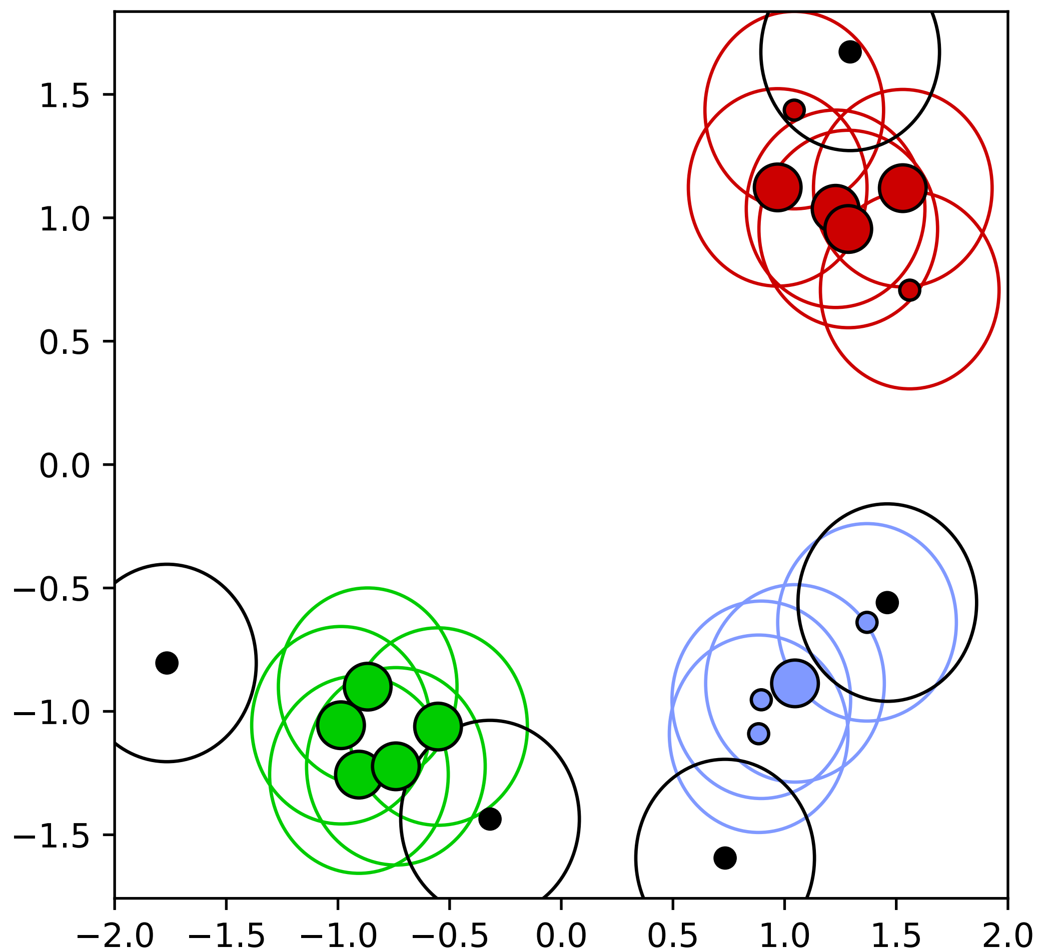


Figure 3.1: Example of a DBSCAN in 2D. The outer circles represents $\epsilon = 0.4$. $minpts = 4$. The black points represents noise, the big points represents core points and all points with same color belong to the same cluster.

3.4 Optical Flow

To track a single cluster of points, optical flow is used on the clusters created with DBSCAN. By tracking the cluster between different frames the process of labeling the points is nearly automatic. Once a cluster has been chosen it can be tracked without the need of any human re-calibration (as long as the cluster is still visible and in the scene). The optical flow algorithm needs two consecutive frames and a few points to track in the first frame as input. It will output new estimated coordinates for the tracked points in the second frame. The following section which describes how optical flow works is based on the original article [11].

For a full explanation of the algorithm used please refer to that article.

Optical flow works by calculating the movement vectors, u and v, which describes the movement for the points in x and y-direction. For optical flow to work it is assumed that the intensity of every tracked pixel remains the same between frames. The translation from one frame to another can then be expressed as the following:

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t) \tag{3.1}$$

Where δx is the translation in x-direction, δy is the translation in y-direction and δt is the time difference of the frames. Using Taylor Series Approximation on the RHS of (3.1) and removing all common terms one will end up with the following expression:

$$\begin{aligned} I(x + \delta x, y + \delta y, t + \delta t) &= I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + ... \\ &\Rightarrow \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t = 0 \end{aligned} \tag{3.2}$$

Finally divide with dt to derive the optical flow equation:

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} \tag{3.3}$$

Where $u = dx/dt$ and $v = dy/dt$. $dI/dx, dI/dy,$ and dI/dt are the image gradients along the horizontal axis, the vertical axis, and time. However solving both u and v given only the equation (3.3) is impossible. A common solution to the problem is using the Lucas-Kanade algorithm. For it to work another assumption needs to be fullfilled; between two consecutive frames the tracked objects are not displaced significantly. The Lucas-Kanade algorithm works by taking a window, normally but not necessarily, small, in this example 3x3. Assuming that all points in the window have the same motion it can be represented in the following way:

$$\begin{aligned} I_x(q_1)V_x + I_y(q_1)V_y &= -I_t(q_1) \\ I_x(q_2)V_x + I_y(q_2)V_y &= -I_t(q_2) \\ &\vdots \\ I_x(q_n)V_x + I_y(q_n)V_y &= -I_t(q_n) \end{aligned} \tag{3.4}$$

Where $q_1, q_2, ..., q_n$ denotes the pixels inside the window and $I_x(q_i), I_y(q_i)$ denotes the partial derivatives of image I with respect to position (x, y) and time t, for pixel q_i at the current time. Now there is an over-determined set of equation for solving V_x and V_y which are the previous u and v vectors. It can be solved by rewriting the set of equation in matrix form, $Av = b$ and using least squares fitting.

Optical flow Lucas-Kanade algorithm works well for this project since LiDAR captures 10 frames per seconds and the objective is to annotate humans which even when running easily will be captured without significant displacements between consecutive frames. During the annotation process all pixels detected have the same constant maximum intensity. An external library for different types of image processing called OpenCV was used to implement optical flow. The pseduo-code for the tracking implemented in this project looks like this:

Algorithm 3.3: Track.

```
1  Track(clusters, frame)
2      newclusters=empty list
3      foreach cluster c in clusters
4          cin2d=convert3dto2d(c)
5          newpoints=opencv.opticalflow(cin2d, frame)
6          newclusters.add(convert2dto3d(newpoints))
7      end
8      return newclusters
9  end
```

Where *clusters* is the list of clusters that were calculated using DBSCAN. Note that the coordinates for these clusters are the true 3D coordinates of the clusters, however the frames that optical flow uses are images, screen shots taken from a birds-eye view of the scene as can be seen in Figure 3.2. The image coordinates is $(0, 0)$ at the upper left corner for the birds-eye view images while for the 3D view which can be seen in Figure 3.3 $(0, 0, 0)$ is located where the LiDAR is positioned which would be represented in the middle bottom of the birds-eye view. Therefore the cluster coordinates need to be translated to the image coordinates using a function called *convert3dto2d* and similarly they need to be translated back to their true 3D coordinates before returning them. The z-coordinates will get lost when translating it back from 2d but the x,y screen shot coordinates are translated to the x,y scene coordinates. It is then possible to recover the z-coordinates because in the main program a new DBSCAN will have occurred for $frame + 1$ and it will map the *newclusters* to the cluster which fits best with these x,y coordinates. Since DBSCAN is preformed independently for every frame they will always get new ids. Therefore it is important to maintain the old id of the cluster so that when the matching with the new clusters is done the new ids also maintain consistent ids. It is then simple to also match labels which is the ultimate goal of the program. The main program for annotating can if simplified be summarized with the pseudo-code in algorithm 3.4.

Algorithm 3.4: Annotater.

```
1  main()
2      frames=loadAllFrames()
3      i=0
4      for i < frames.length -1
5          clusters=dbScan(frames[i])
6          2dClusters = Track(clusters, frames[i])
7          newClusters = dbScan(frames[i+1])
8          match(2dClusters, newClusters)
9          autoAnnotate(newClusters)
10         saveDataToFile()
11 end
```

Where *match* is the necessary function to match the 2D coordinates given from optical flow with the new clusters generated from dbscan. *autoAnnotate* checks what the clusters previously was labeled as and gives that labels to the new matched clusters. So the labeled clusters in $frame$ gets matched with the new clusters in $frame + 1$ and automatically receives the same label. Finally the data is saved as a csv file in *saveDataToFile* which can easily be used for machine learning.

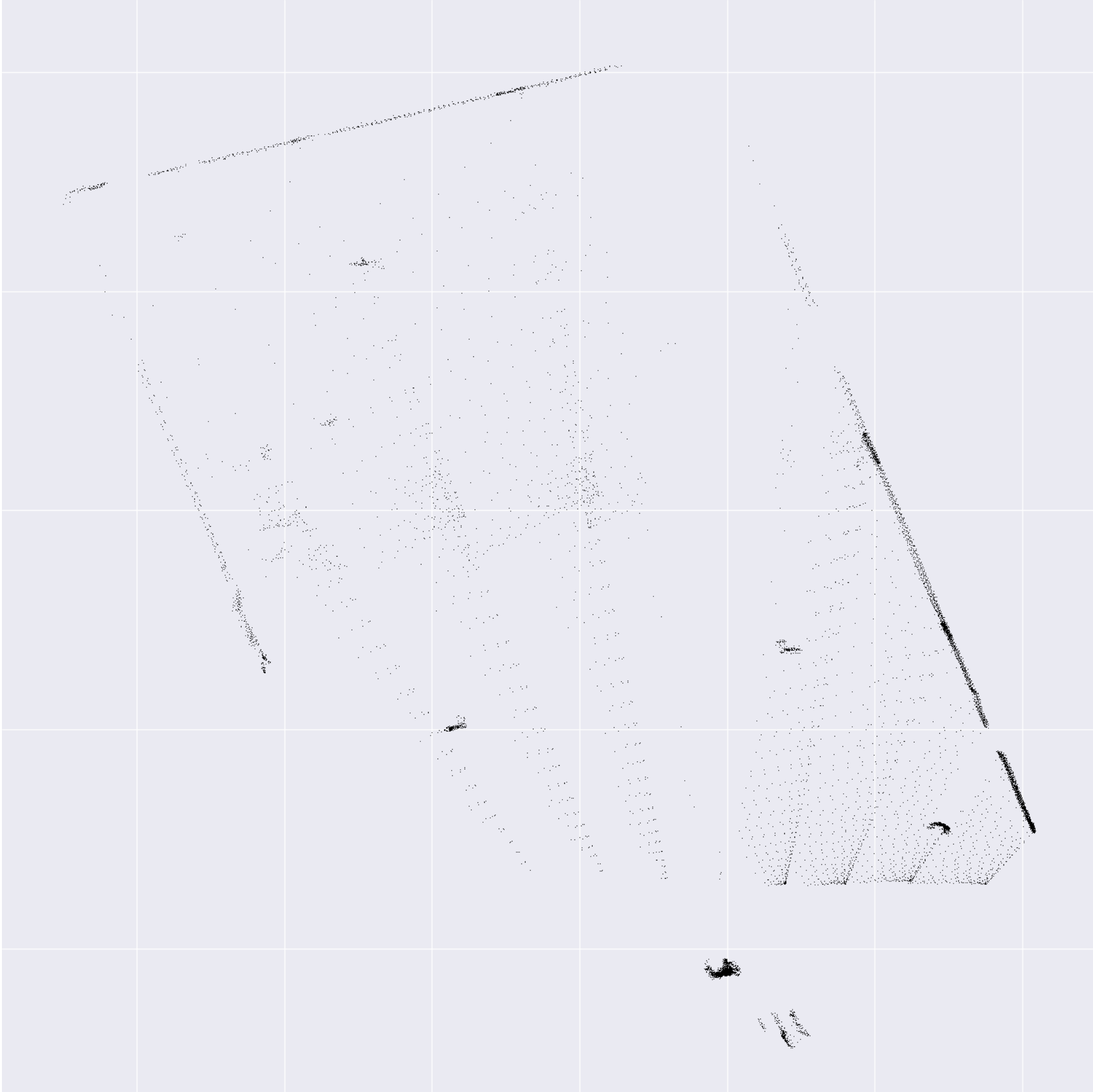


Figure 3.2: The birdseye view of the scene which is the view optical flow operates on. The human is located in square (5,1) in the bottom right corner.

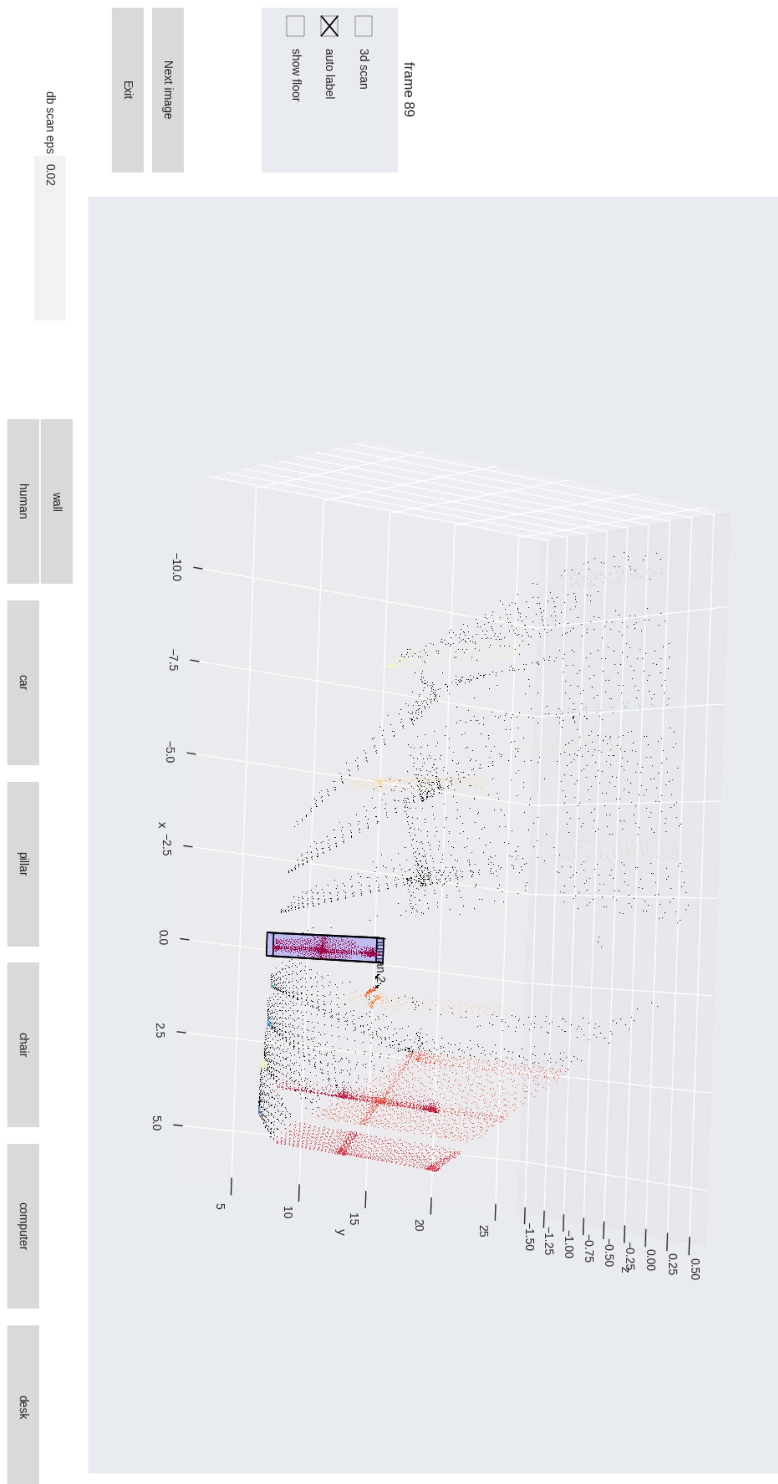


Figure 3.3: The 3D view of the scene where a human has been labeled. The scale and zoom of the two images are not an exact fit. But they are both from the exact same frame. Image has been rotated for better scale.

Figure 3.3 is the view the operator of the annotator views. The buttons are different categories that objects can be labeled as. Those buttons are fully customizable in a config file however it is important to maintain consistent labeling between data sets if they are to be used to train a model. There is also the possibility to change ϵ for DBSCAN to get better clusters.

3.5 Sweeping Annotator

When using the clustering annotator many of the clusters are much larger than the ones which will be found during actual use of the classifier. Therefore a sweeping annotator was implemented, this annotator takes a point cloud which contains no humans and scans a box through the point cloud. The box with the x, y dimensions of 0.7x0.7m (and a height that is adjusted to the points contained withing the x and y limits) around the scene in steps of 0.7m. The points contained in the box are clustered with DBSCAN on the x and z values of the points, point clouds that are larger than 20 points are labeled and stored to be used as training and testing data. This means that the clusters are smaller than 0.7x0.7m and are at least 20 points large. A similar process is used during actual use of the algorithm to find clusters that potentially are human (see 4.2.2).

Chapter 4
Classification

4.1 Training process

4.1.1 Overview of the Training Process

The training process starts once enough data has been annotated. There are two major steps of the training process; feature extraction and training of models. The features are information extracted from clusters and is what the classifiers are trained on to distinguish between different classes. In this project the classifiers are trained on voxel features as well as a few other features inspired from [4]. Once the features are extracted they are stacked in a vector, stored as a csv file and are finally used for training of the classifiers.

Three different classifiers were tested; Random Forest (see 4.1.6), Support Vector Machine (see 4.1.7) and K-Nearest Neighbors (see 4.1.8), for all classifiers different parameters are compared to find the optimal solution for this project. The strength and weaknesses of the classifiers are then compared to decide which classifier is the most effective and well suited for this project.

4.1.2 Voxel Grid

To get a sized vector containing some information of the shape of an object it is common to use some kind of voxel method (such as [6] and [5]). A grid of voxels (voxel grid) is a volume which has been divided into smaller volumes, each of these smaller volumes (voxels) represents a feature which is to be used for the classifier. The value of the feature depends on whether the voxel is occupied or not, unoccupied voxels corresponds to 0 when viewed as a feature and 1 when occupied. These features are then stacked as a vector. The voxel grid is an excellent method to get some sort of edge detection of 3D point clouds. A visualization of a voxel grid containing a human can be seen in Figure 4.2. The number of voxels can be varied, a higher number of voxels increases the resolution but it also increases the computation time.

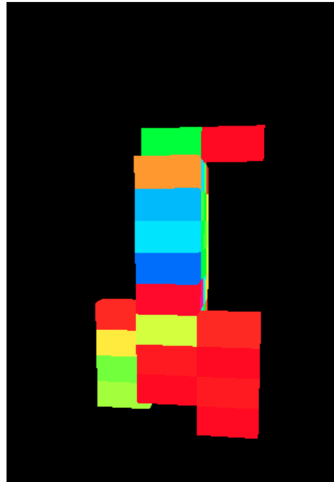


Figure 4.1: A voxel grid of a human cluster seen in profile, made of 10x5x10 voxels, the color of the voxels correspond to how many points are in the voxel.

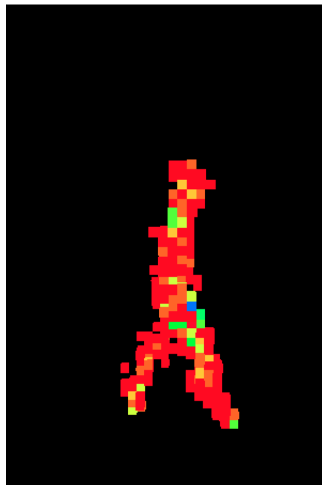


Figure 4.2: A voxel grid of the same human cluster as in Figure 4.1 seen in profile, made of 30x30x30 voxels.

In this project each voxel grid is 500 (10x5x10) voxels large.

4.1.3 Features

In addition to the voxel features, 11 other features are extracted and added to the feature vector. Some of the features (Equation 4.1-4.10) used are based on the ones used in [4],

Mean Intensity

$$f_1 = \frac{\sum_{i=1}^{n} I_i}{n} \tag{4.1}$$

where n is the number of points and I_i is the intensity of point i

Width and Height

$$f_2 = \Delta x = x_{max} - x_{min} \tag{4.2}$$

where x_{min} and x_{max} is the minimum and maximum x-values of the point in the cluster

$$f_3 = \Delta y = y_{max} - y_{min} \tag{4.3}$$

where y_{min} and y_{max} is the minimum and maximum y-values of the point in the cluster

$$f_4 = \Delta z = z_{max} - z_{min} \tag{4.4}$$

where z_{min} and z_{max} is the minimum and maximum z-values of the point in the cluster

Number of points

$$f_5 = n \cdot y_{min} \tag{4.5}$$

PCA Ratio f_6 : The Ration between the two largest singular values received from singular value decomposition of the points in a cluster.

Area of bounding box

$$f_7 = 2 \cdot (\Delta x \cdot \Delta y + \Delta x \cdot \Delta z + \Delta y \cdot \Delta z) \tag{4.6}$$

Normalized Cartesian Dimension

$$f_8 = \sqrt{\Delta x^2 + \Delta y^2} \tag{4.7}$$

Centroids in bounding box

$$f_9 = \frac{\sum_{i=1}^{n} x_i}{n} - x_{min} \tag{4.8}$$

$$f_{10} = \frac{\sum_{i=1}^{n} y_i}{n} - y_{min} \tag{4.9}$$

$$f_{11} = \frac{\sum_{i=1}^{n} z_i}{n} - z_{min} \tag{4.10}$$

4.1.4 The Feature Vector

By stacking the voxel and the other features (500+11) the result is a 1x511 features long vector for each cluster to be sent to the classifier. Two separate files containing the feature vectors are created, one for clusters created by the clustering algorithm and one from the sweeping algorithm. The reason for creating two separate files is for convenience, the process of extracting features from a cluster is time consuming and if the objective of recreating the feature vector is to add more data it is much simpler to do so for either objects or humans separately. The training dataset contains close to 25 500 non-human and 10 500 human clusters.

4.1.5 Classifiers

The choice of classifier is not arbitrary as they differ in strength and weaknesses. Variables to consider when choosing classifier are amount of data, features, whether the data is labeled or unlabeled, computation time and what type of data is used. During this project a comparison was made between Support Vector Machines (see 4.1.7), Random Forest (see 4.1.6) and K-Nearest Neighbors (see 4.1.8).

The objects that are tested is acquired by using the sweeping annotator (see 3.5) in different environments. The test dataset was gathered from different environments to get diversity in the type of clusters. Data was gathered both in- and outdoors. None of the testing data was captured at the same location as any of the training data, this is done to minimize correlation between the training and testing data. The test dataset was created using the same sensor as was used for the gathering of training data.

4.1.6 Random Forest

A decision tree is a predictive model which constructed as a tree of nodes and leaves, each node is a test on an feature and is split into two other nodes. The test that is performed chooses how to split the node the best way, this is done by having the Gini Index criterion selecting a test that maximizes 4.11. This process is repeated until the the Gini impurity equals zero and nodes are no longer split, these last nodes are called leaves and the class of the leaf is the predicted label of the item to be classified. When training a decision tree 100% accuracy is always achieved on the training data (i.e the tree can successfully classify all items in the training data).

Whenever a node is split in two the aim is to make the new nodes purer than the parent node. The most common ways of achieving this is by applying either the Gini Impurity or Information Gain functions. They are used as a way to estimate which test of the features yields the best split. [12] argue Gini gain and information gain yield very similar results empirically and they also concluded that they differ in only 2% and that it is therefore not possible to decide which function yields the best results. In this project only the Gini gain was used.

$$gini(T) = 1 - \sum_{i=1}^{k}(p(c_i))^2 - \sum_{i=1}^{k}(p(t_i)) \sum_{j=1}^{k}(p(c_j|t_i)(1 - p(c_j|t_i)) \qquad (4.11)$$

Where:

t is the outcome of the test

k is the number of classes (two for a binary classifier)

c is the class

Ho proposed in [13] that a forest of trees should be used instead of a single tree, a forest is an ensemble method which means that it uses multiple learning algorithms (multiple decision trees) to achieve better result. The Random Forest classifier consists of more than one tree where each node is trained on a subset of randomly selected features. When the using the Random Forest to classify an item, the item is classified by all trees and each tree casts a vote as to which class the belongs to. The result of the vote is used as the score from the forest and is for a binary classifier the ratio of votes, e.g. $[.9, .1]$ means that 90% of the trees classify the item as class 0 and 10% as class 1.

When this classifier was implemented the variables that were varied were the number of trees and the number of random features to select when training the decision trees. In general Random Forest is not a very time consuming method but the time needed to train the classifier is increased as the number of trees increase. Random Forest is a good choice for high dimension data (many features) and it is usually very resistant to overfitting.

4.1.7 Support Vector Machine

This basic explanation of support vector machines is based on the article [14] and for the full details of support vector machines including all the maths behind it please refer to that article.

A Support Vector Machine (SVM) tries to classify items by adding a hyperplane that splits the items into different classes. It works well for labeled data which is called supervised learning. For a simple 2D problem like the one shown in Figure 4.3a the hyperplane can easily separate the two classes with a simple line. But most problems are not as simple, for example the problem presented in Figure 4.3b is unsolvable by using a line. However if an extra dimension is added, like in Figure 4.3c it is again an easily solvable problem by adding a plane. Going back to the original 2D space the desired result is achieved as can be seen in figure 4.3d.

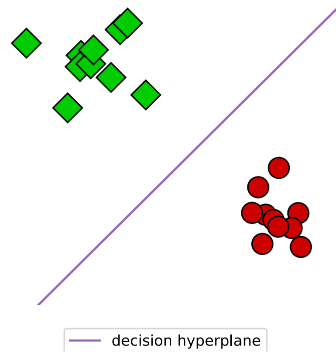
Normally the problem is not as straightforward and it is common to find some overlap between the two classes as can be seen in Figure 4.4a. These problems are also solvable with SVM in the same way as mentioned above however the designer of the SVM will have to choose if one wants to solve the problem perfectly as in Figure 4.4b or allow some outliers as in Figure 4.4c. Normally one would allow some outliers to fit the general problem better and not overfit to the training data. That problem is not exclusive for SVMs, it is common in most machine learning algorithms. The parameter that regulates how well to fit to the data points is called regularization.

A problem with SVMs as described so far is that it is very computationally heavy to do the mapping from points in one dimension to points in higher dimensions and then find the best hyper plane that splits the data. To reduce the computations something called kernels are used. To explain how kernels work start by taking note that the optimal hyper plane will be the hyperplane that has the largest margin between the hyperplane and the decision boundaries which can be seen graphically in Figure 4.4d. The problem can be expressed as

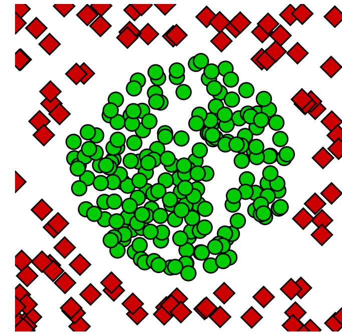
a constrained optimization problem, constrained by the support vectors and optimization because one wants to maximize the margins. It can be shown that the constrained optimization problem solely depends on the dot product between the feature vectors in the SVM, that is $x \cdot y$ where x and y are feature vectors. A kernel take advantage of this fact and is simply a function that maintains the dot product properties, that is $K(x, y) = \phi(x) \cdot \phi(y)$ where K is the kernel function. Since the kernel maintains the properties of the dot product there is no longer a need to calculate the dot product, instead one can apply the kernel function to the data. The beauty of all this is that there is no longer any need to do any explicit mapping between dimensions in space, instead one will use the kernel functions to solve the constrained optimization problem. This mathematical trick is often referred to as the kernel trick. Choosing the correct kernel for the problem is an important parameter, some of the most commonly used kernels are Linear, Polynomial and Radial Basis Function (RBF) kernels. The other two parameters that needs to be set are γ (Equation 4.13) and a margin parameter which normally is denoted as C. The γ regulates for every training example which data points should be considered when calculating a separation line. A high gamma results in only the closest data points to the plausible separation line is considered while a low gamma results in data points far away are also considered. The margin parameter regulates how far the separation line should be from the support vectors, that is the data points closest to the line (see Figure 4.4d).

$$K(x, y) = exp(-\frac{||x - y||^2}{2\sigma^2}) \tag{4.12}$$

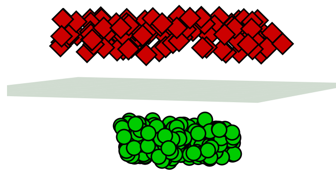
$$\gamma = \frac{1}{2\sigma^2} \tag{4.13}$$



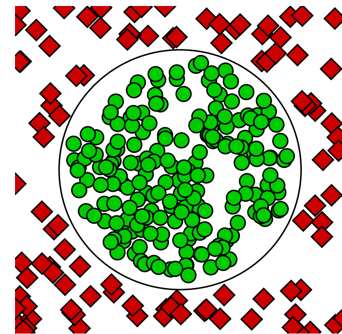
(a) Example of a linear problem that is easily solvable by SVM.



(b) Example of a non-linear problem that is easily solvable by SVM.

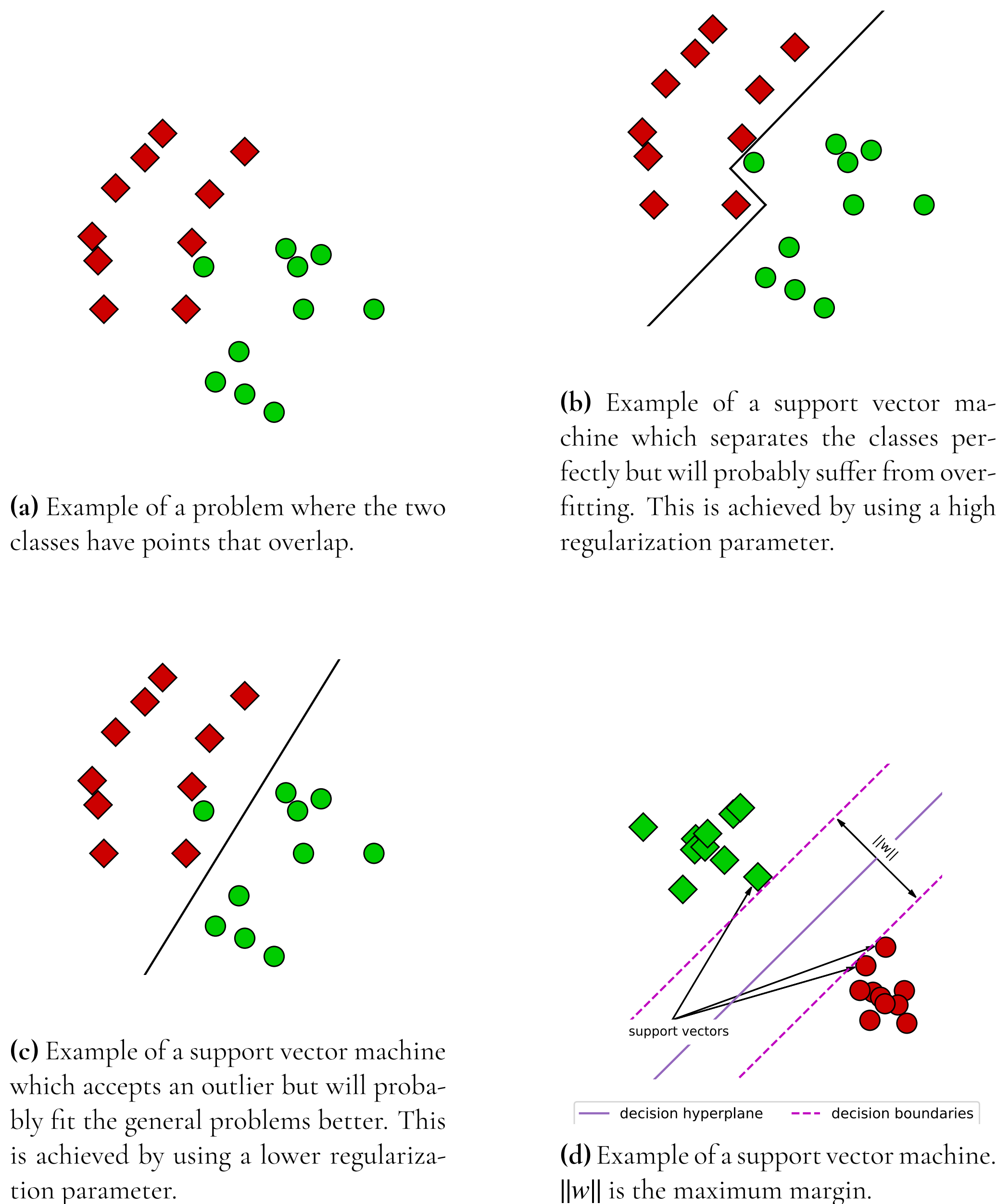


(c) Example of a SVM which successfully classified the two classes.



(d) Example of a SVM where the feature space is in one higher dimension than the input space which makes it possible to separate the classes with a plane.

Figure 4.3: Illustration of various SVM problems.



(a) Example of a problem where the two classes have points that overlap.

(b) Example of a support vector machine which separates the classes perfectly but will probably suffer from overfitting. This is achieved by using a high regularization parameter.

(c) Example of a support vector machine which accepts an outlier but will probably fit the general problems better. This is achieved by using a lower regularization parameter.

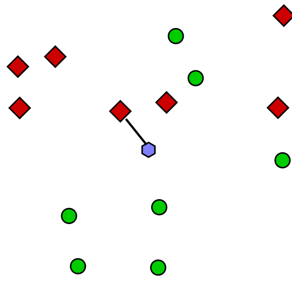
(d) Example of a support vector machine. $||w||$ is the maximum margin.

Figure 4.4: Illustration of various SVM problems.

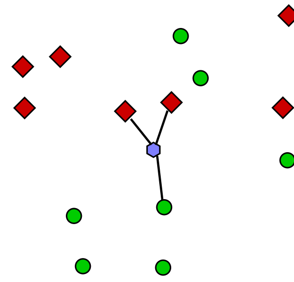
4.1.8 K-Nearest Neighbors

K-Nearest neighbors classifier is one of the most simple classifiers in machine learning and has been around for quite some time [15]. For any unlabeled point it will check its k nearest neighbors and be labeled to the class which has the majority among the neighbors. k can be chosen but is normally small which can make the classifier very noisy. To reduce the noisy data one can chose a larger k however that will result in less clear boundaries between the classes. When considering a larger k value one normally weights the neighbor to favor the neighbors closer to the point [16]. To determine the distance between points a distance function needs

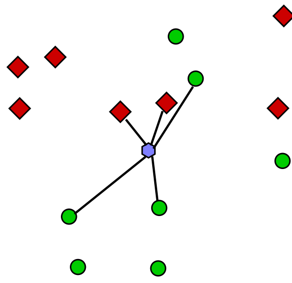
to be defined. For continuous variables that function is normally euclidean distance however other distance functions can sometimes work better and for discrete variables which can be found e.g. in text classification other functions has to be chosen. k should optimally be chosen so that there can not be an even number of neighbors with different classes, e.g. in a problem with two classes chose k as any positive odd number. However that is not always possible when there are more than 2 classes, in these cases ties can occur and it is up to the designer how to resolve it. Another variety of K-Nearest Neighbors is to not define k but instead define a radius where all points within the radius of the unlabeled point is considered. The labeled points are then weighted depending on the distance from the unlabeled point. A normal weight factor is $\frac{1}{d}$ where d is the distance from the unlabeled point to the labeled point. In this case ties are very unlikely but still not impossible. A great advantage with K-Nearest Neighbors is that it is easy to understand and also easy to implement, the big drawback is that there are a lot of distance calculations which can become very slow in large data sets. In figure 4.5a, 4.5b and 4.5c one can see how the neighbors are chosen for $k = 1, k = 3$ and $k = 5$ respectively when euclidean distance is used. The result of the labeling of the hexagon will be red diamond, red diamond and green circle for $k = 1, k = 3$ and $k = 5$. In 4.5d one can see that if $radius = 1$ it will get labeled as red diamond. When using $radius = 2$ it is not as obvious what the resulting label would become. Using the weighting function $\frac{1}{d}$ the result will be $\frac{1}{0.86} + \frac{1}{0.92} = 2.243$ for red diamond and $\frac{1}{1.56} + \frac{1}{1.12} + \frac{1}{1.89} = 2.065$ for the green circles which again would result in a red diamond labeling when the green circles actually have majority.



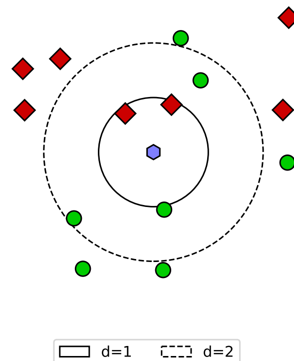
(a) Example of K-Nearest Neighbors where $k = 1$. The blue hexagon is the unknown point that are to be labeled.



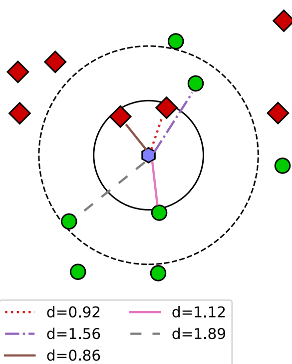
(b) Example of K-Nearest Neighbors where $k = 3$. The blue hexagon is the unknown point that are to be labeled.



(c) Example of K-Nearest Neighbors where $k = 5$. The blue hexagon is the unknown point that are to be labeled.



(d) Example of K-Nearest Neighbors using different radius. As can be seen in this example when the radius is one two points are considered while when the radius is two 5 points are considered.



(e) The distances to the neighbours within a radius of 2 are displayed.

Figure 4.5: Illustration of K-Nearest Neighbors.

4.2 Classification Process

4.2.1 Overview of the Classification Process

After training the model it is now possible to test the model on new data. After capturing some data using the LiDAR the pcap file is fed directly to the algorithm. In each frame a box is being swept throughout the point cloud (just as described in 3.5). Each cluster in the box is saved in an array of point clouds and is fed to the classifier once the whole frame has been searched. A bounding box around clusters that are classified as human are created, if there are multiple boxes on a single object (see Figure 4.6) non-maximum suppression is used to remove the ones that fit the cluster the least.

Unlike in the annotation process the classification is fed a pcap file (the raw data) recorded with the LiDAR, the data needed for the algorithm is extracted from this by using code from the SDK.

4.2.2 Identifying Interesting Clusters

The process of finding clusters that potentially are human resembles a lot the sweeping annotator (see 3.5), a box of size 0.7x0.7m is moved along the x-axis with a step-size of 0.3m, when it reaches x_{max} the box is moved 0.3m along the y-axis and the box is again moved along the x-axis. The points in the box are discarded if to box contains contains less than 80 points. The time complexity of the whole classification algorithm is largely dependant on the step-size of the box, as the step-size decreases the amount of clusters increase.

Inside of the bounding boxes are usually multiple clusters of points, if data is collected indoors it is then common that there are parts of the ceiling and floor within each box. These clusters are separated using a two- dimensional version of DBSCAN (see 3.3) where the point cloud is projected on a plane along the y-axis, if a cluster contains more than 20 points the cluster is relevant and is sent to the classifier.

To further decrease the time complexity, a hard limit of 30m is set for the y-axis, clusters farther away are usually too sparse to be successfully classified. This hard limit also substantially decrease the time spent on larger point clouds. If a LiDAR with higher accuracy was used then the hard limit of 30m could be increased.

4.2.3 Prediction and Non-Maximum Suppression

The prediction is done by feeding a cluster to the trained model that was created during the training process (see 4.1). The classifier returns the probability of each label. Since the step size is rather small and the bounding box is large the same cluster can be sent to the classifier multiple times in different boxes. Since the same cluster can be contained in multiple sweeping boxes there is a high probability that a cluster will be classified as human multiple times. This is shown as multiple bounding boxes overlapping (see Figure 4.6), to remove these superfluous boxes Non-Maximum Suppression (NMS) is used. NMS iterates through an array of bounding boxes and removes boxes that overlap more than a set threshold leaving only one bounding box around each cluster. The decision of which bounding box is kept depends

on the score (probability) received from the classifier, the box around the cluster with the highest score of being a human is the one that is kept (see Figure 4.7).

The NMS algorithm that was implemented is usually used on pictures instead of point clouds. Since the boxes are being moved along the x- and y-axis, it would be appropriate to only use the x and y coordinates. The overlapping boxes instead become overlapping squares on the z-plane and the two dimensional version of NMS is applied on the squares to decide which one to use. The algorithm receives an array of the corners to all squares and returns the corners of the square with the best fit.

The NMS algorithm used is based on the NMS implemented in [17].

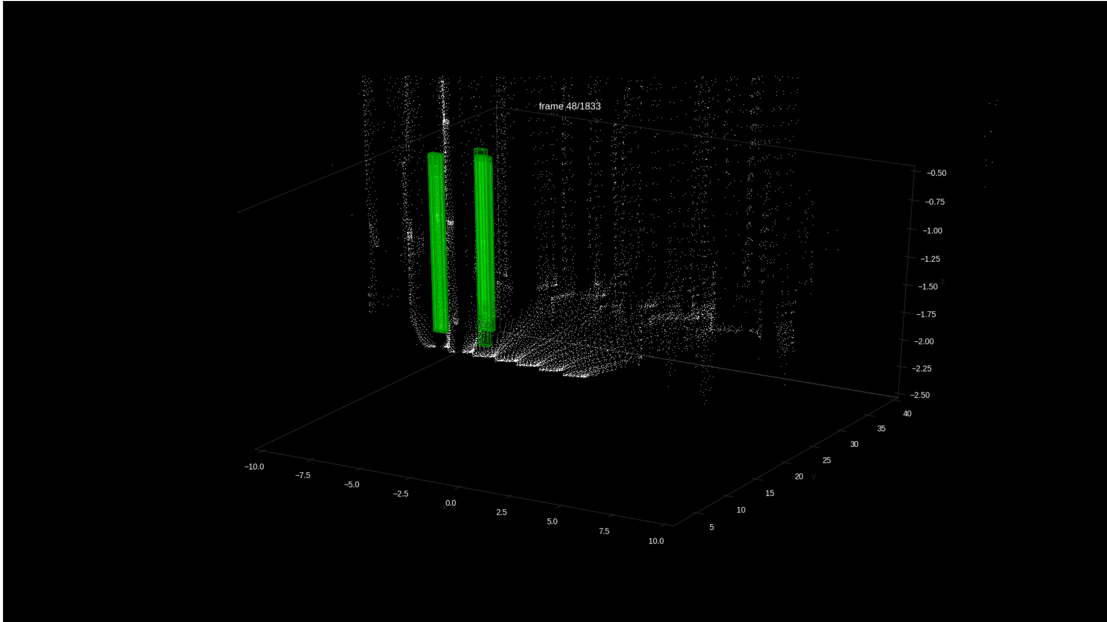


Figure 4.6: Two humans which have been classified in multiple bounding boxes, no NMS is used.

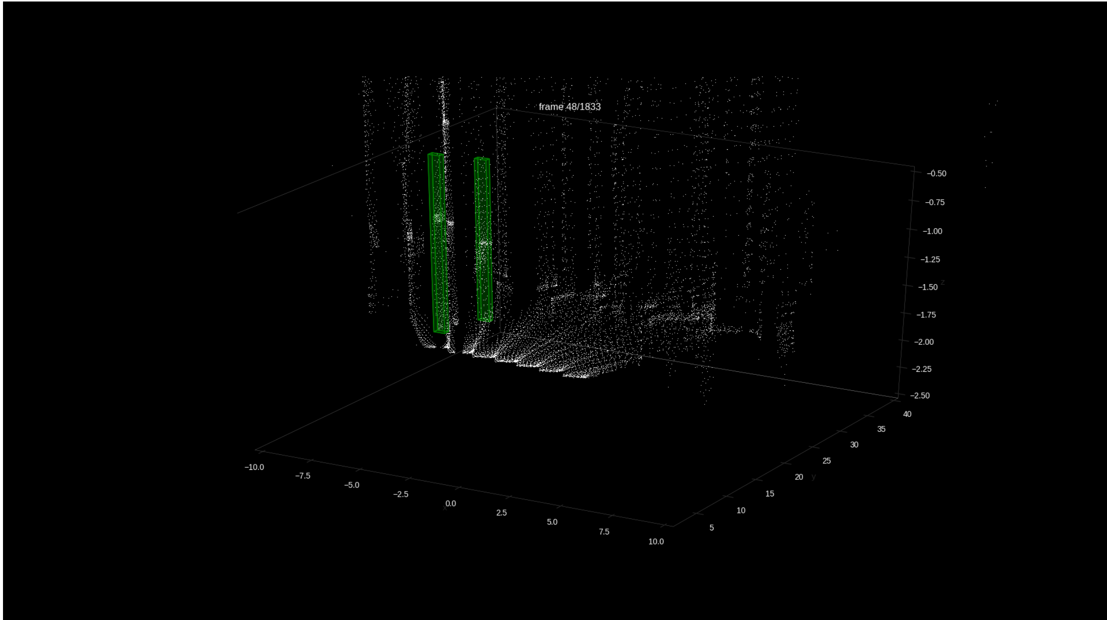


Figure 4.7: The same two humans as in Figure 4.6 but with NMS, notice that there is only one bounding box per cluster.

4.3 Evaluation of classifier

To evaluate the classifier a test dataset was created containing 1 700 human (around 30 different humans) and 16 200 object clusters. The data in the training dataset is highly correlated as the data was gathered from consequent frames. Therefore the test dataset was gathered from other locations and of other pedestrians, to get the data as divers as possible when it comes to clusters. Data was gathered in both inside and outside environments. The test dataset was created using the same sensor as was used for the gathering of training data.

When comparing different binary classifiers it is usually a good idea to look at the Receiver Operating Characteristic (ROC) curves. A ROC curve shows the True Positive Rate (TPR) (defined as 4.14) plotted against the False Positive Rate (FPR) (defined as 4.15). Most often when creating a ROC-plot, multiple plots are plotted in the same graph as a mean to compare either different classifier with each other or compare a classifier while varying a parameter. Figure 4.8 is an example of the ROC curve of one well performing classifier and one with worse performance. The diagonal line is the result of a random guess. The Area Under Curve (AUC) (defined as 4.16) corresponds to the accuracy, if AUC $= 1$ the accuracy is 100%. A classifier that performs well will be close to the upper left corner of the graph and have a high AUC.

$$TPR = \frac{TP}{TP + FN} \tag{4.14}$$

$$FPR = \frac{FP}{FP + TN} \tag{4.15}$$

$$AUC = \int_{x=0}^{1} TPR(FPR^{-1}(x))dx \tag{4.16}$$

Where:

TP: True positive

FN: False negative

FP: False positive

TN: True negative

AUC: Area under curve

The training and prediction time of the test dataset was compared to give an idea of how much the classifiers differ and to decide which classifier is the most effective if the accuracy is similar. The training time is not very significant but it is interesting to have some kind of idea of how long each algorithm need to be trained.

The following values of the classifiers were compared using ROC curves;

Random Forest: Number of features used in each tree ($\log_2 n$, $\sqrt{n}$ and n) and the number of trees (10,20,50,100 and 1000)

Support Vector Machine: With RBF as kernel the value of γ was varied ($10^{-5}, 3 \cdot 10^{-5}, 7 \cdot 10^{-5}, 10^{-4}, 3 \cdot 10^{-4}, 7 \cdot 10^{-4}, 10^{-3}, 10^{-2}, 0.1$)

K-Nearest Neighbors: The number of neighbors (3, 5, 10, 50, 100)

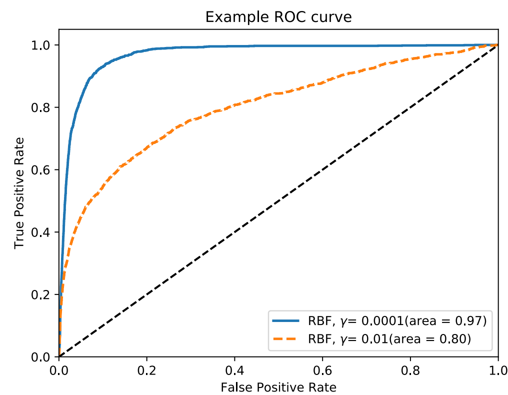


Figure 4.8: Example of ROC curves, the two curved lines corresponds to two classifiers with different parameters. The curved dotted line has a worse performance than the solid one. The straight diagonal dotted line corresponds to a line where the outcome of the classifier is random (like flipping a coin).

Lastly an evaluation of the features is done. The importance of each feature is plotted for the best performing classifier, this shows, when combined with the duration of the feature extraction what features are reasonable to use in a real time implementation.

Chapter 5
Results

5.1 The Sensor

Overall the sensor has worked very well and is, thanks to the SDK very easy to use. The LiDAR technology is very interesting and rather fun to test out.

When looking through glass the sensor does not always work as well as expected, some appear to be semi-transparent with some points going through the glass and some are reflected. There is also a significant difference in intensity for points on glass. Using the LiDAR with the intention of looking through glass would not be advised.

There seems to be some issue with some frames where frame can have significantly less points in one frame whereas the next frame contains significantly more points (see Figure 5.1). As there are so few points it's impossible to do any classification but since it is only one frame it can usually be overlooked. This is most likely only a problem for this LiDAR model.

It is possible for a human viewing footage to follow a person walking away from the LiDAR for abut 80m as there are still some points returned at this distance. If a person instead walks toward the camera it is much more difficult to decide whether the moving object actually is a human and for a definite decision the person needs to be about 20-25m from the LiDAR.

The LiDAR seems to be unaffected by strong sun and works well in slight rain.

The overlap between the different channels has posed some problems when using the classifier, quite often false positives are created in the overlapping areas.

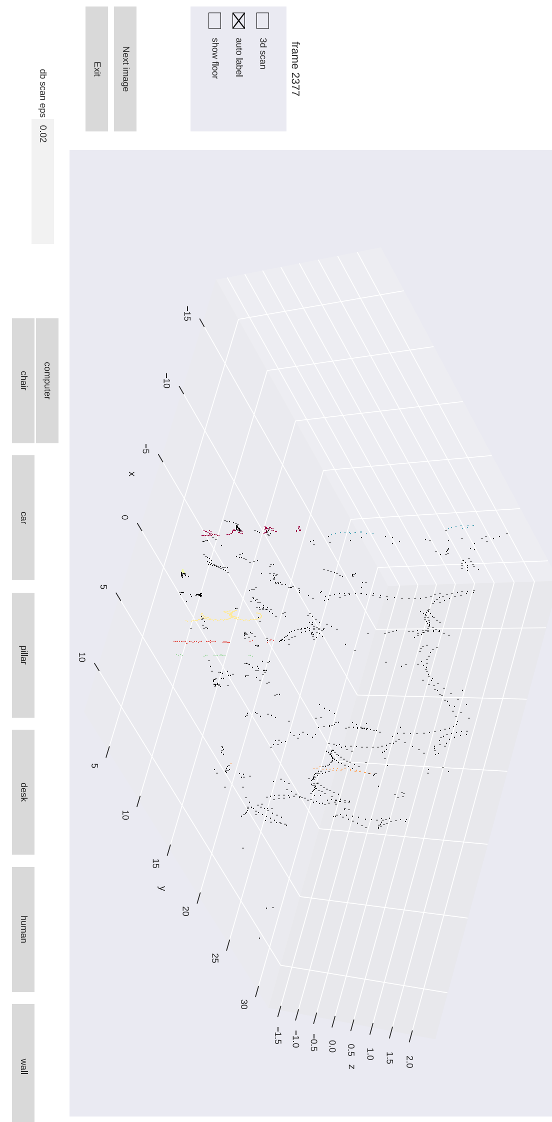


Figure 5.1: A frame from the sensor with lower resolution, occasionally occurs when using the LiDAR.

5.2 Plane Removal

The resulting GUI for removing planes can be seen in Figure 5.2. Although the GUI is a bit unresponsive it worked well for the scenes we used it for. However, we did only do about 30 scenes where the majority were indoors scenes.

To the right are the boundaries in x,y and z-axis. In Figure 5.2 everything is within boundaries so everything is marked green. In Figure 5.3 the threshold for the maximum height is set to 1.1829 which makes all the blue points out of boundary. This way the operator can clearly see which points are to be removed before actually doing so. The button **find a plane** will try to find a plane anywhere within the green points, this works very well if the boundaries are set appropriately but in the general case when the entire scene is within boundaries it works very poorly. **Find floor** works very well in most scenarios, normally on the first try. The red plane seen in Figure 5.4 was found using the **find floor** button. Problems will occur if the floor is a too steep slope. The plane is marked red so that the operator again clearly can see which points will be deleted. If the plane is a bad one the operator can simply re-click find floor or do a manual plane finding search. There is also the possibility to change the parameters for RANSAC but the default values have been proven to be a good trade off for accuracy and speed in the scenes we have tried.

The GUI is very unresponsive which demands that the operator has very high patience, this is because the python package used for plotting, Matplotlib, seems to have trouble drawing so many points. It can take several seconds for changing any boundaries parameters or changing the viewing angle. Figure 5.5 shows the final result after removing both blue points and the red plane, as can be seen from this top view image it is a very clean scene where only pillars walls and a few objects are left. This is the result one should strive for to get optimal performance when annotating data later.

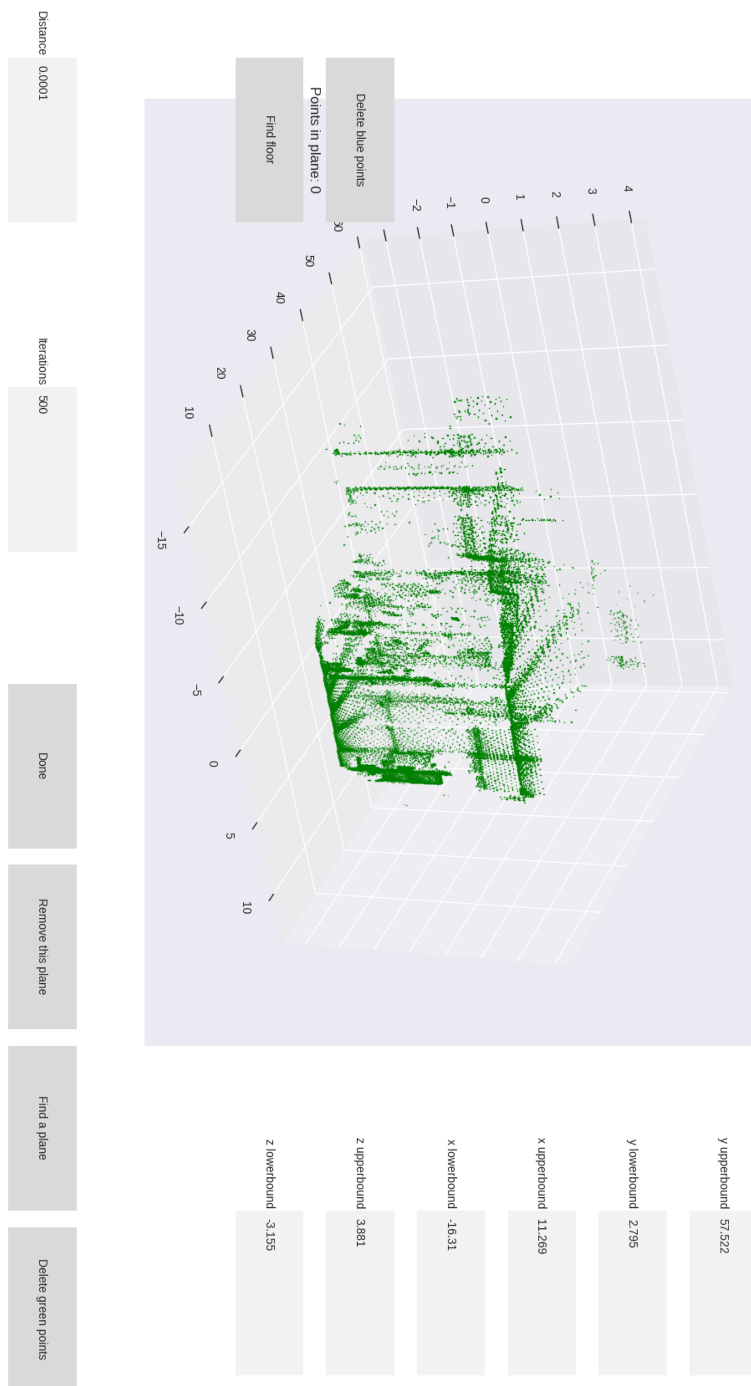


Figure 5.2: Overview of the GUI for removing planes.

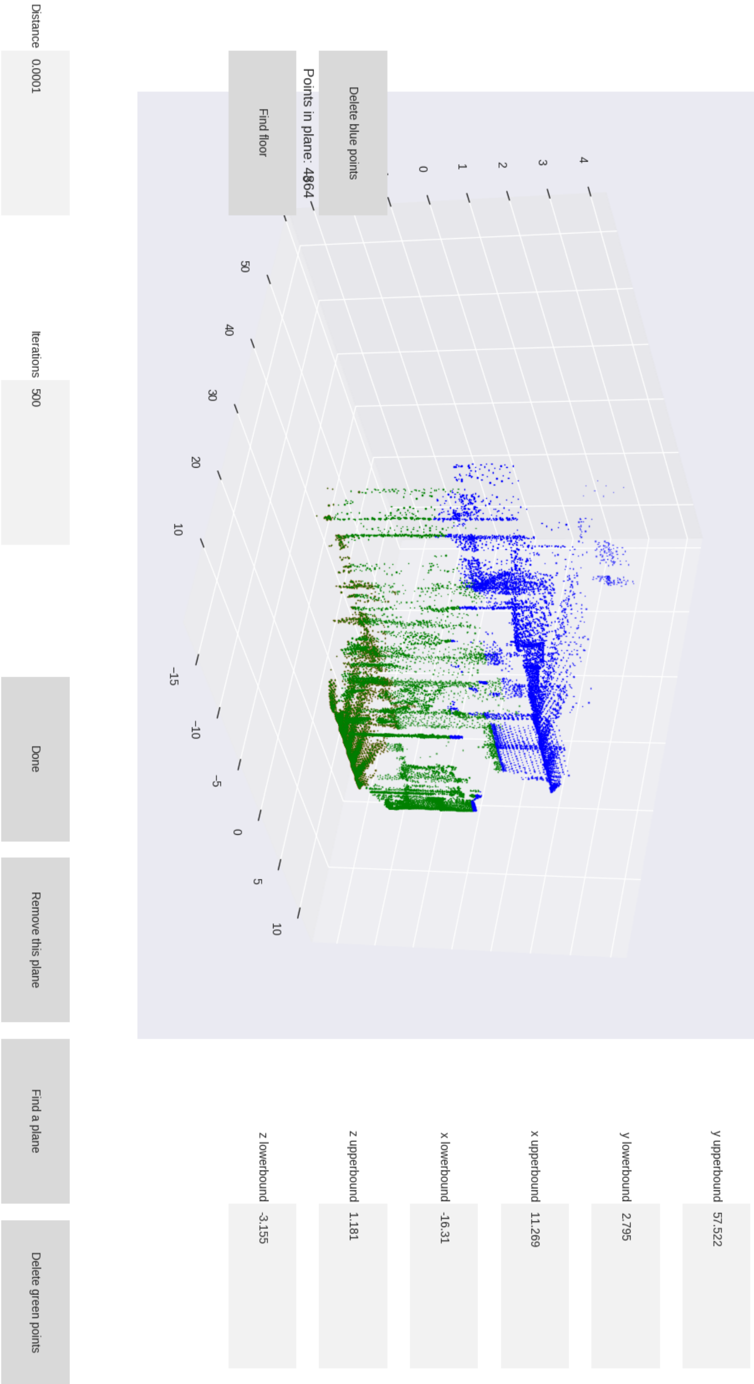


Figure 5.3: The blue points are points outside of boundaries and considered for deletion.

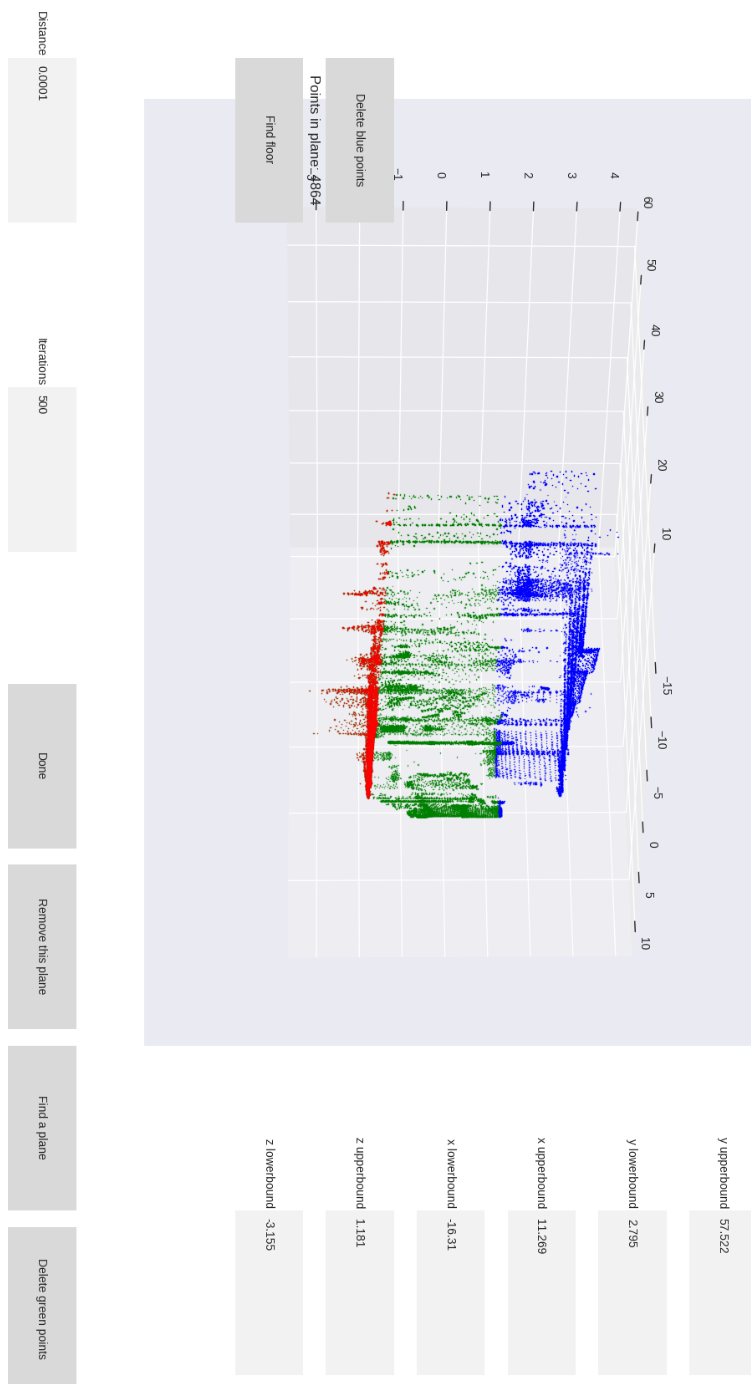


Figure 5.4: This is the same scene from a different angle which is slightly below the scene. The red points are the plane found by RANSAC and for consideration for deletion.

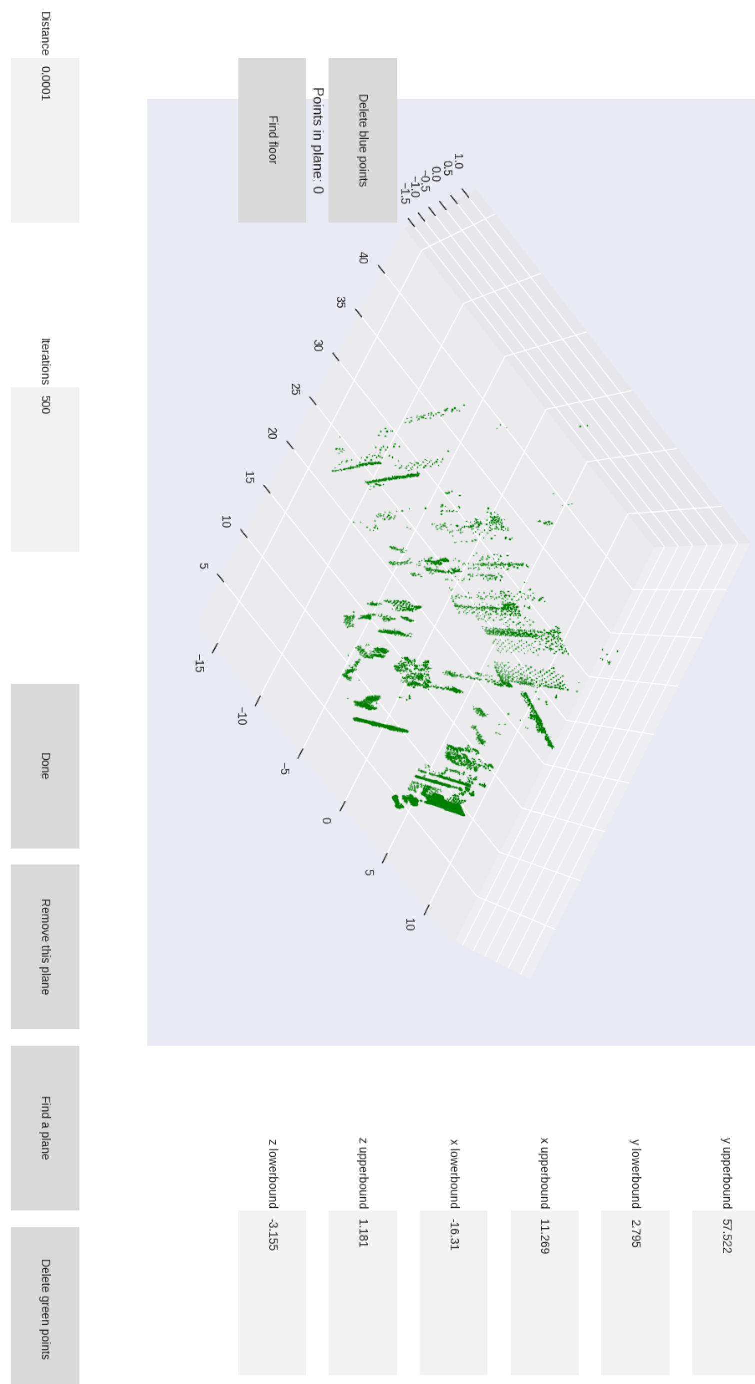


Figure 5.5: Final outcome of the floor removal and ceiling removal. It is the same scene viewed from a top angle.

5.3 Annotator

The final GUI of our annotator looks like the one in Figure 5.6. The annotator works very well within a distance of 1 to 20 meters from the sensor as long as the floor and ceiling has been properly removed and the humans do not walk too close to other objects. Too close being in the range of a few centimeters if close to the sensor or a few decimeters if far away from the sensor. The difference is due to the point density decreases when far away from the sensor and therefor it is not possible to cluster as accurately when far away.

All the different colors in Figure 5.6 represents different clusters, black color represents noise and does not belong to any cluster. If the operator is not satisfied with the current clusters it is possible to change the radius of DBSCAN to get new clusters within the GUI. This is very useful and crucial for getting good clusters that captures only a human and not anything extra or missing part of a foot etc. The Annotator as can be seen has support for labeling several different objects than humans, but in this project we ended up only classifying humans. Once the operator is satisfied with the clusters the labeling starts. When the box is green it shows that is the cluster that is currently being labeled, the blue boxes are clusters that has already been labeled and they have their labels showing on top of the boxes. Objects labeled as "other" are ignored. Once the labeling phase is done for that frame one can go to the next frame. An illustration of the labeling process can be seen in Figures 5.6 and 5.7. Every object that has been labeled to anything else than "other" will get tracked into this next frame. As can be seen in Figure 5.8, there are 21 frames after the frame in Figure 5.7 the human is still accurately tracked while all the objects labeled "other" were ignored. Since people are moving around and sometimes go close to each other or objects the operator will sometimes have to adjust the radius of DBSCAN to only capture the wanted object and nothing else. As long as the floor and ceiling are removed in a similar way as in figure 5.5 this works very well. The main problem is when people go to close to objects and the people clusters become one with the objects. As long as the humans are far enough away from any objects it is possible to annotate 100-200 frames in roughly 10 minutes which is a massive improvement compared to the annotators which we found and tried for annotating LiDAR data. When we tried Hitachi semantic segmentation editor[18] it took us between 1-5 minutes for annotating a single frame. We looked at some other editors too but their method was the same as Hitachi so it did not seem like we could get any great time improvements using theirs. It is reasonable to assume we could annotate faster than 1-5 minutes once getting used to their annotator but it is not very likely that it would become that much faster that it would come close to the speed we annotate using our own program.

Unfortunately the auto-scaling and camera view in Matplotlib made it hard, sometimes impossible for the operator to clearly see what was being annotated.

Figure 5.6: Overview of the final GUI for annotating objects. The green box is the cluster that is currently being labeled which in this case is a human. The blue boxes have already been labeled.

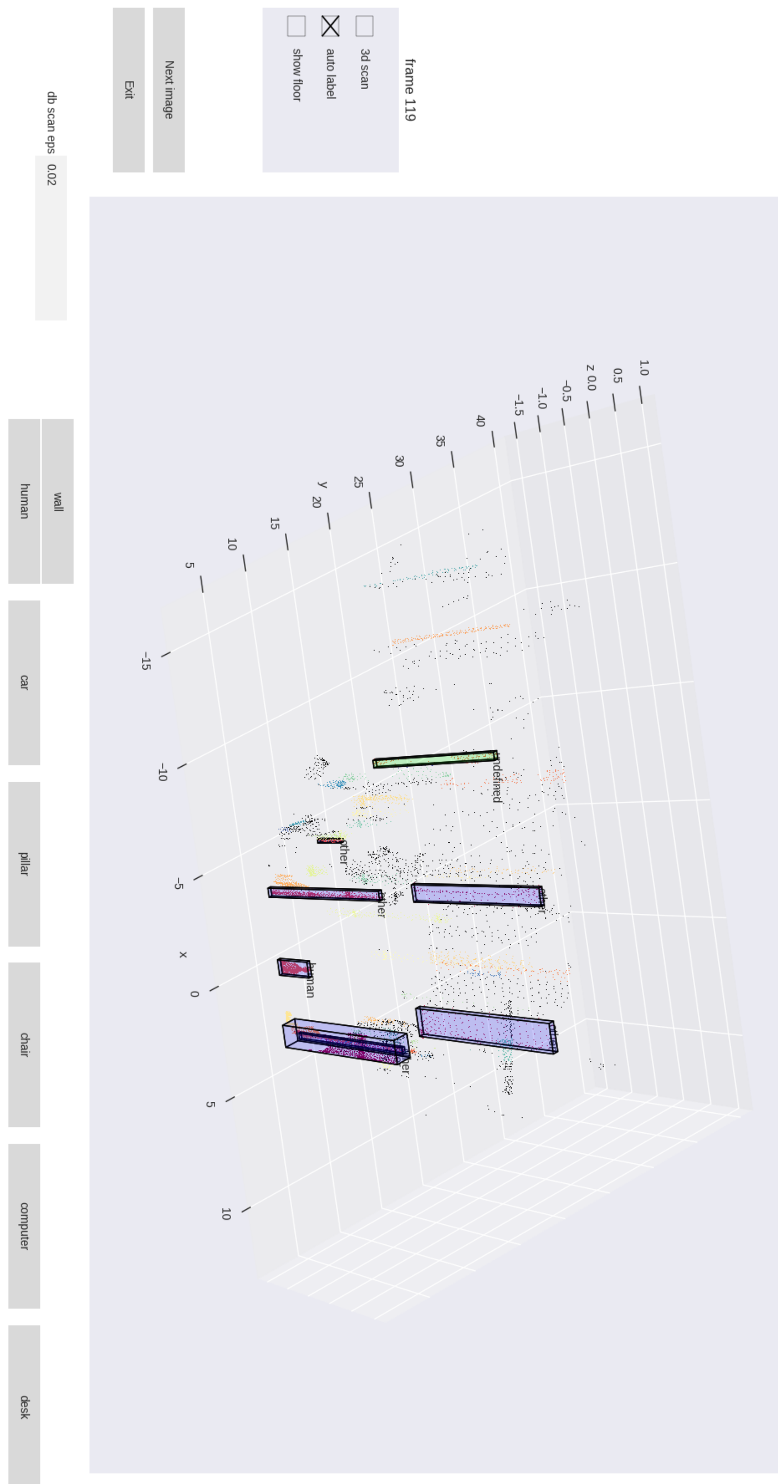


Figure 5.7: The same frame as in figure 5.6 where the operator has labeled the human cluster as a human.

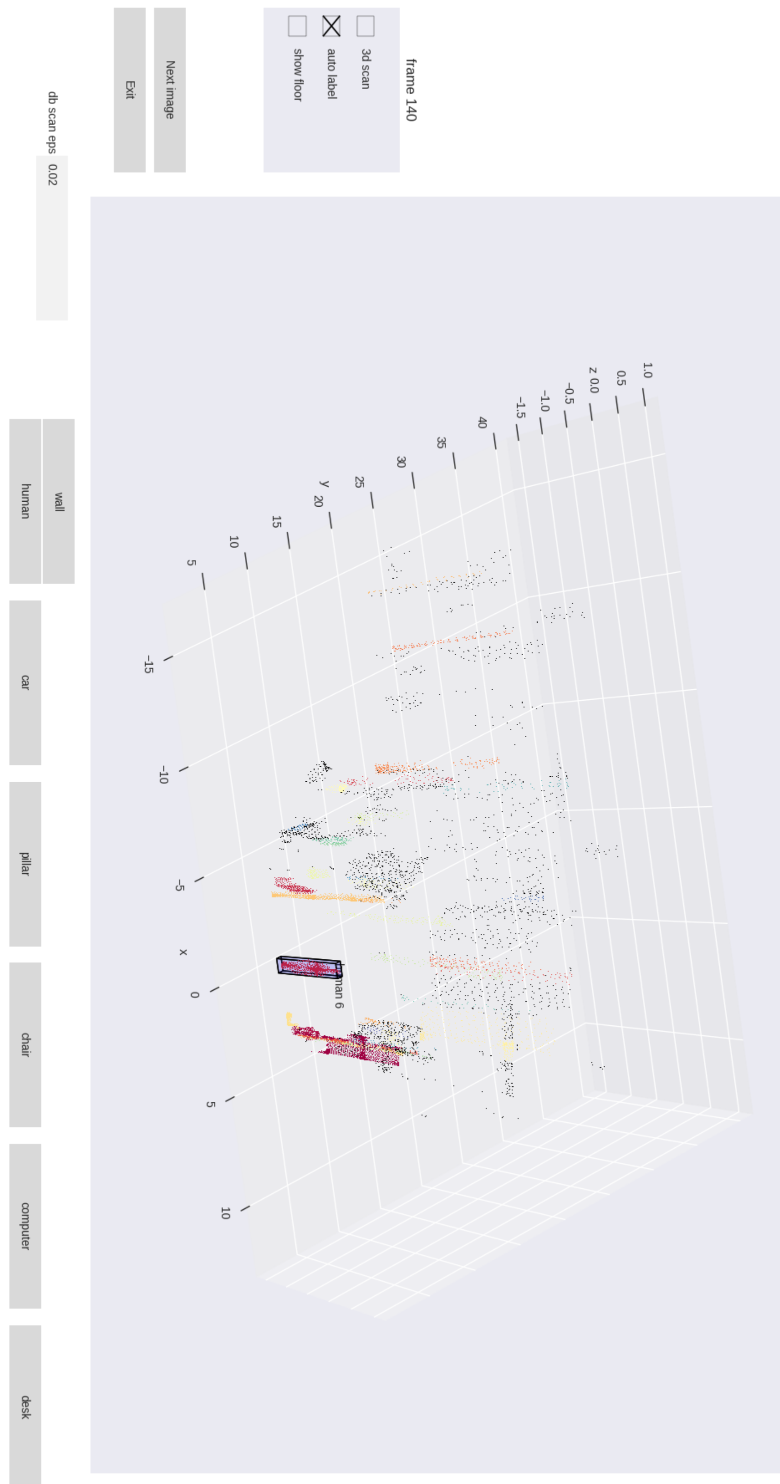


Figure 5.8: 21 frames after the one in figures 5.6 and 5.7 where the human still is correctly tracked. All the operator has to do in this case is click next frame.

5.4 Classifiers

5.4.1 The classifiers in general

All of the classifiers are able to achieve high accuracy but the best performing one in this project is the Random Forest algorithm as it has the highest accuracy and a reasonable prediction time.

5.4.2 Random Forest

As can be seen in Figures 5.9, 5.10 and 5.11 all of the classifiers perform very well and achieves a 99% accuracy even with lower amounts of trees and features used. The choice of which classifier to use then depends on the computation time. From Figure 5.12 it is clear that there is no major difference in classification time for forests smaller than 100 trees, for larger forests it is however faster to use all features in each tree. The downside for forests with trees using all features when training a tree is that there is a significant increase in training duration as can be seen in Figure 5.13. When using the Random Forest algorithm with 1 000 trees, it is possible to detect humans up to 20m from the sensor.

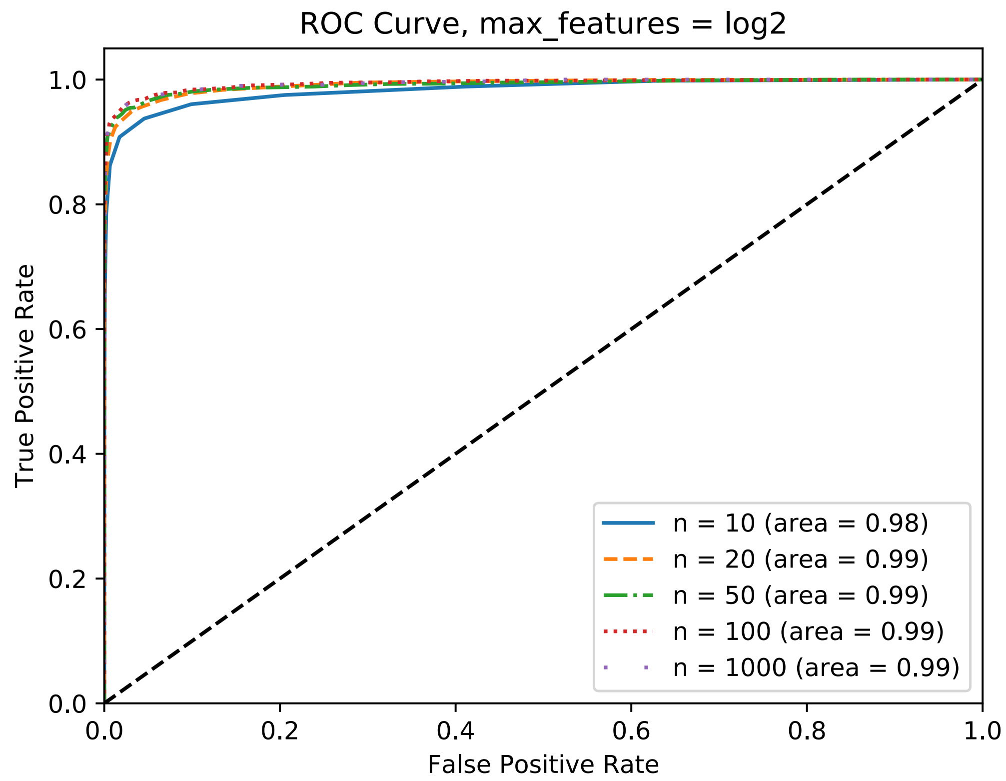


Figure 5.9: ROC curve of Random Forest classifier with $\log_2 n$ features used , each curve is calculated on a different number of trees.

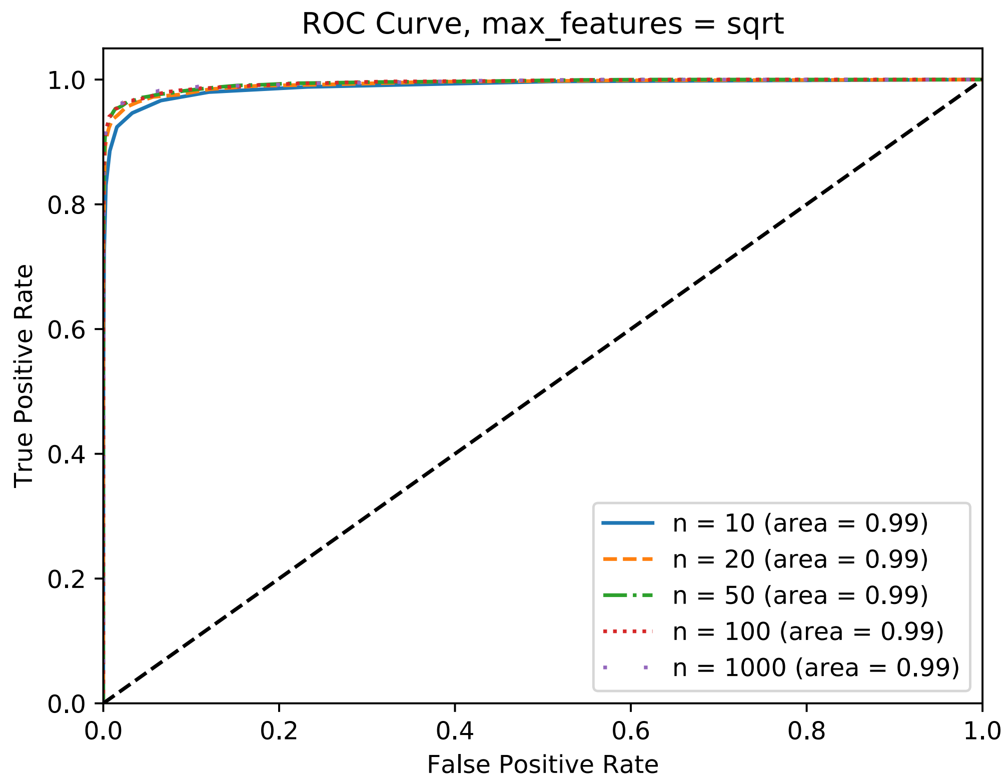


Figure 5.10: ROC curve of Random Forest classifier with $\sqrt{n}$ features used , each curve is calculated on a different number of trees.

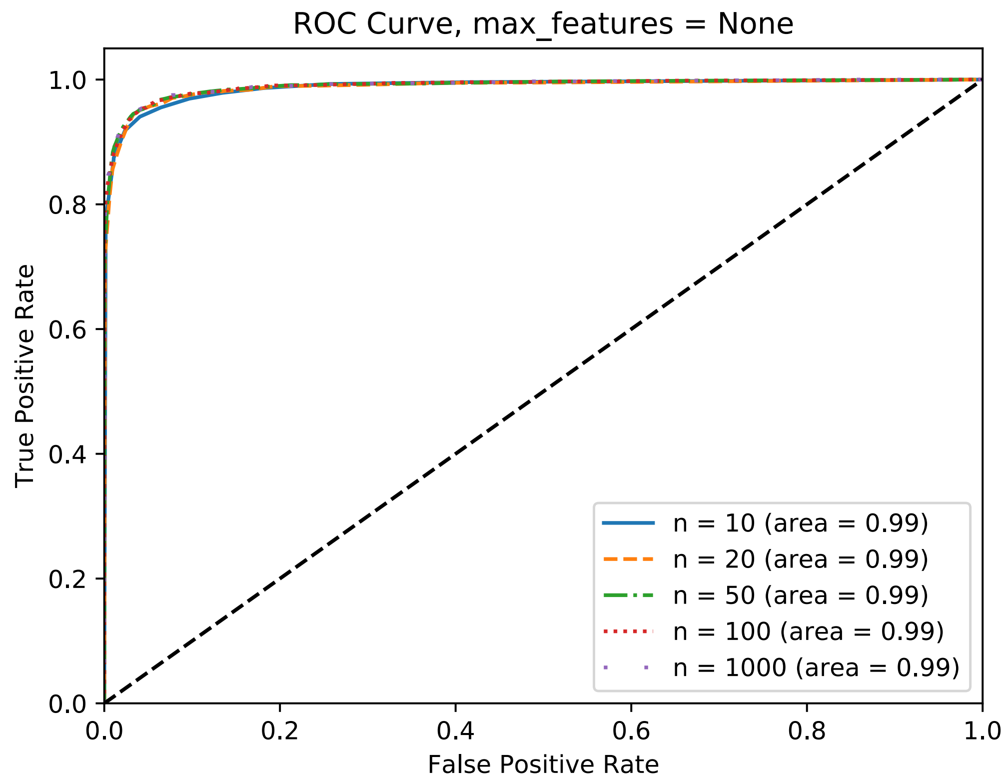


Figure 5.11: ROC curve of Random Forest classifier with n features used, each curve is calculated on a different number of trees.

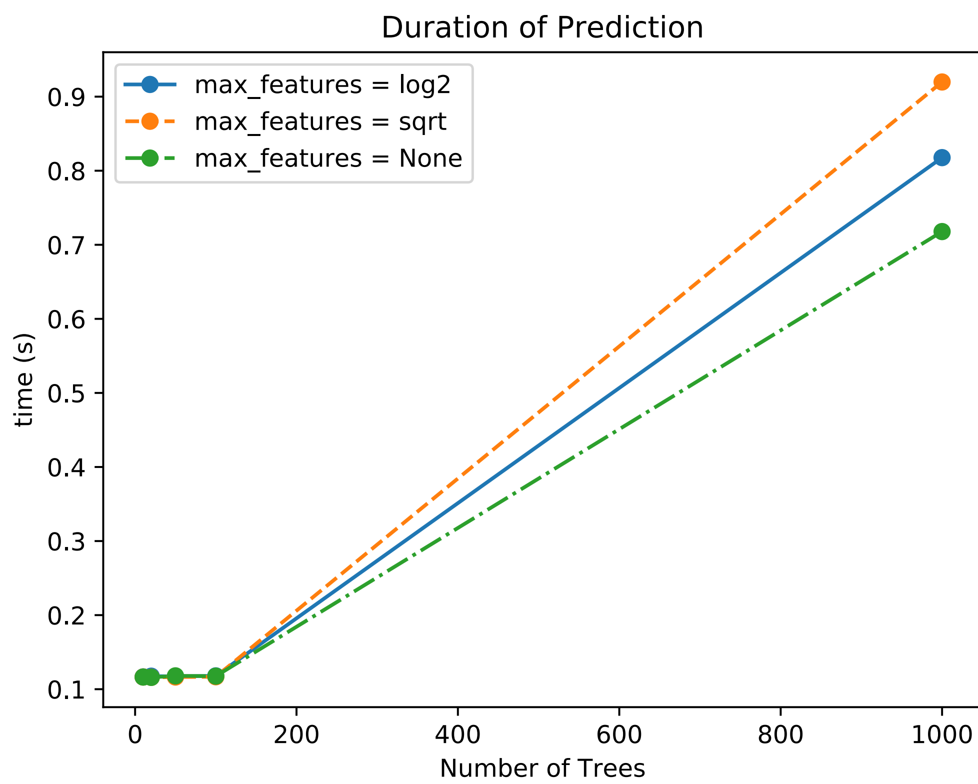


Figure 5.12: Duration of classification using the Random Forest classifiers.

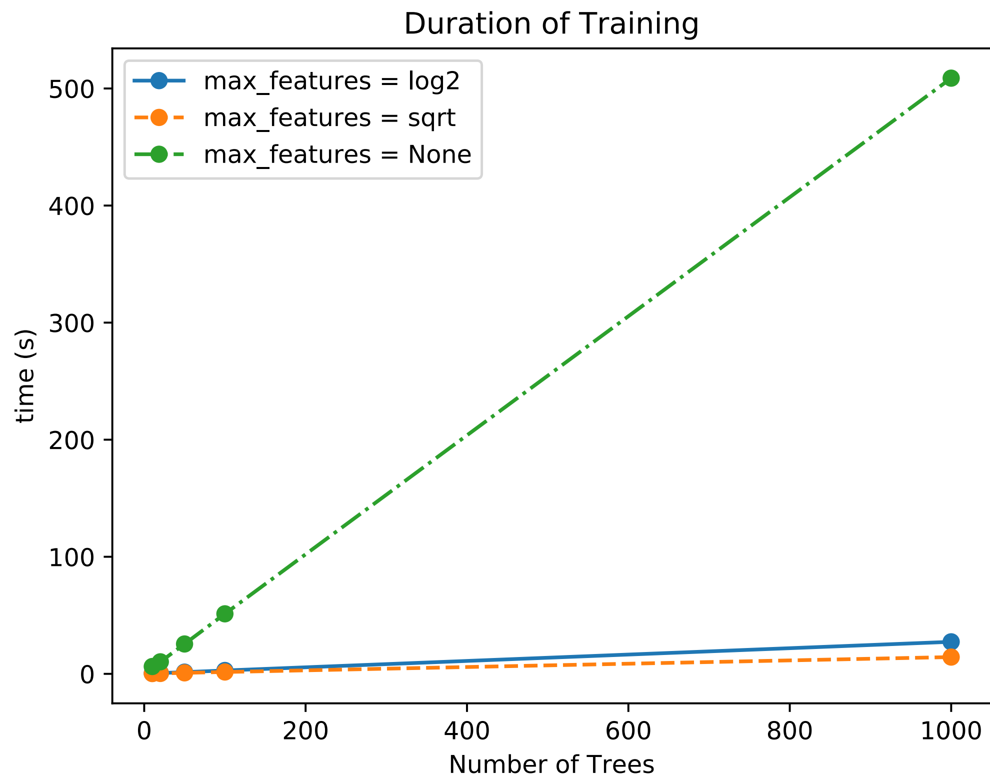


Figure 5.13: Duration of training of the Random Forest models.

5.4.3 Support Vector Machine

The support vector machine classifier performs very well when it comes to accuracy for some values of γ, the performance does not get better for lower values but there seems to be an optimal value around 10^{-4}. However, no better performing value was found. The duration of training and predictions in Figure 5.16 and 5.17 seem to vary a lot between different values of γ however there is only a minor difference in time and it varies more between different training and prediction sessions than with each other.

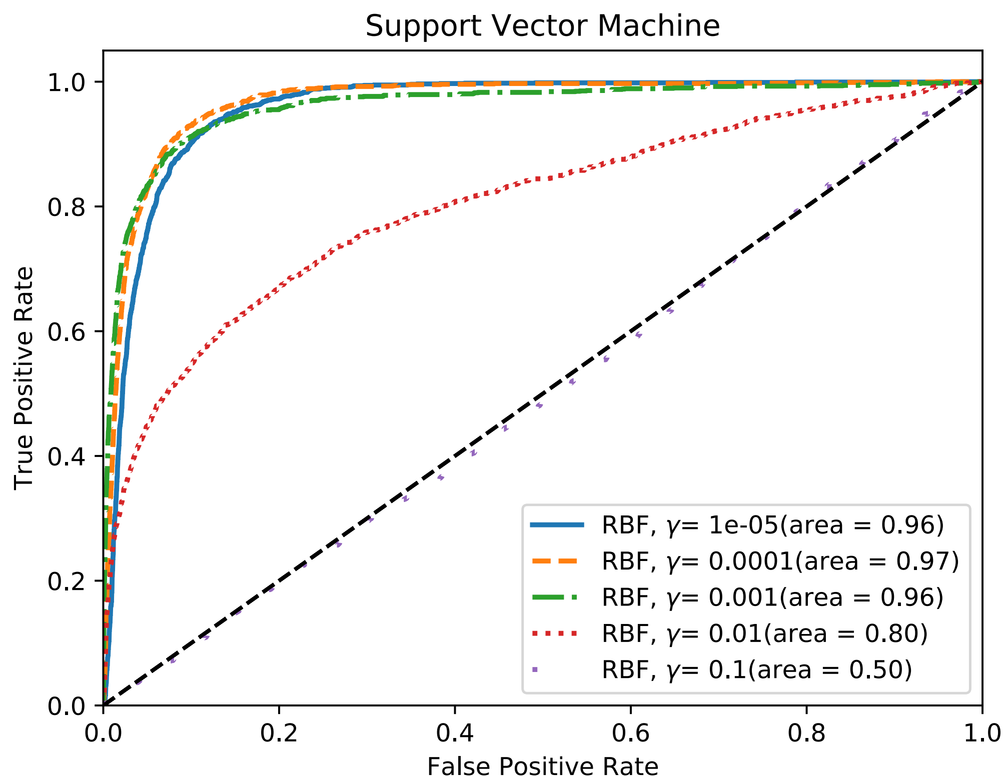


Figure 5.14: ROC curve of a support vector machine where the kernel is radial basis function and each curve is calculated using a different value of gamma.

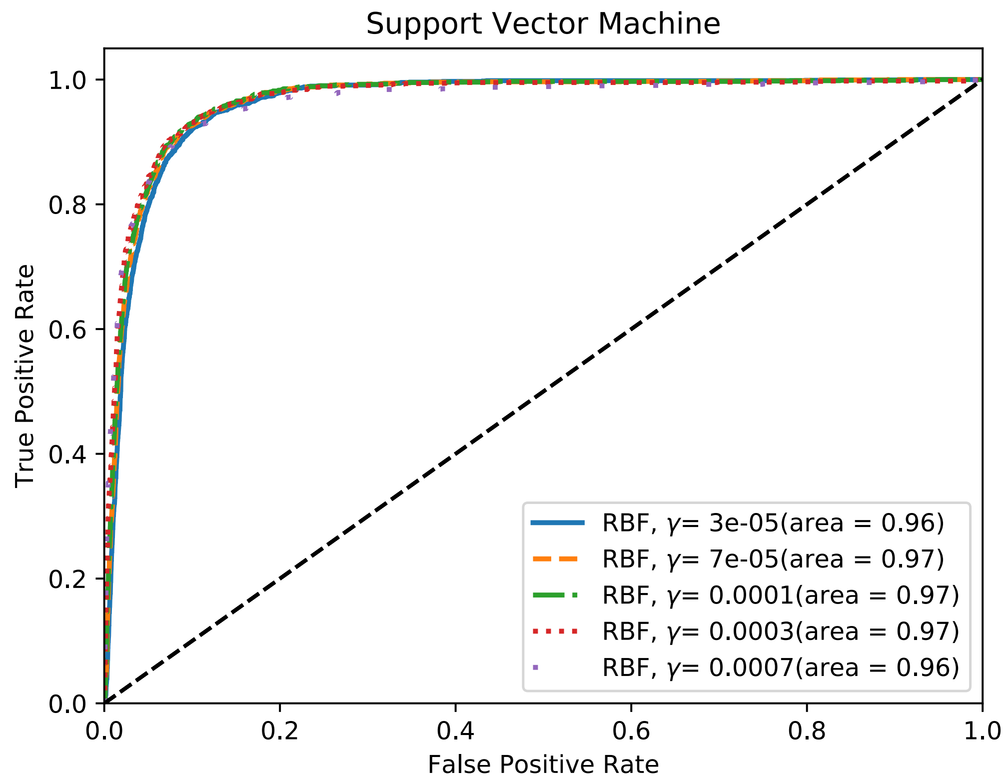


Figure 5.15: ROC curve of a support vector machine where the kernel is radial basis function, an attempt to find a better value of γ was performed.

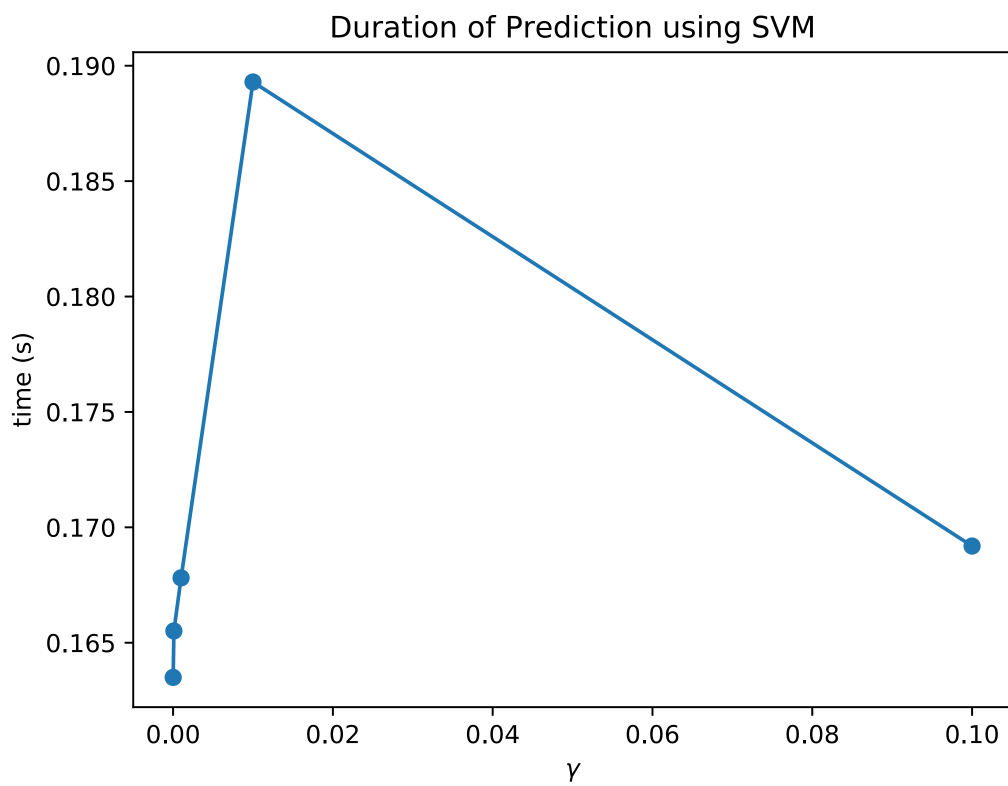


Figure 5.16: Duration of classification using SVM with maximum iterations = 10^3.

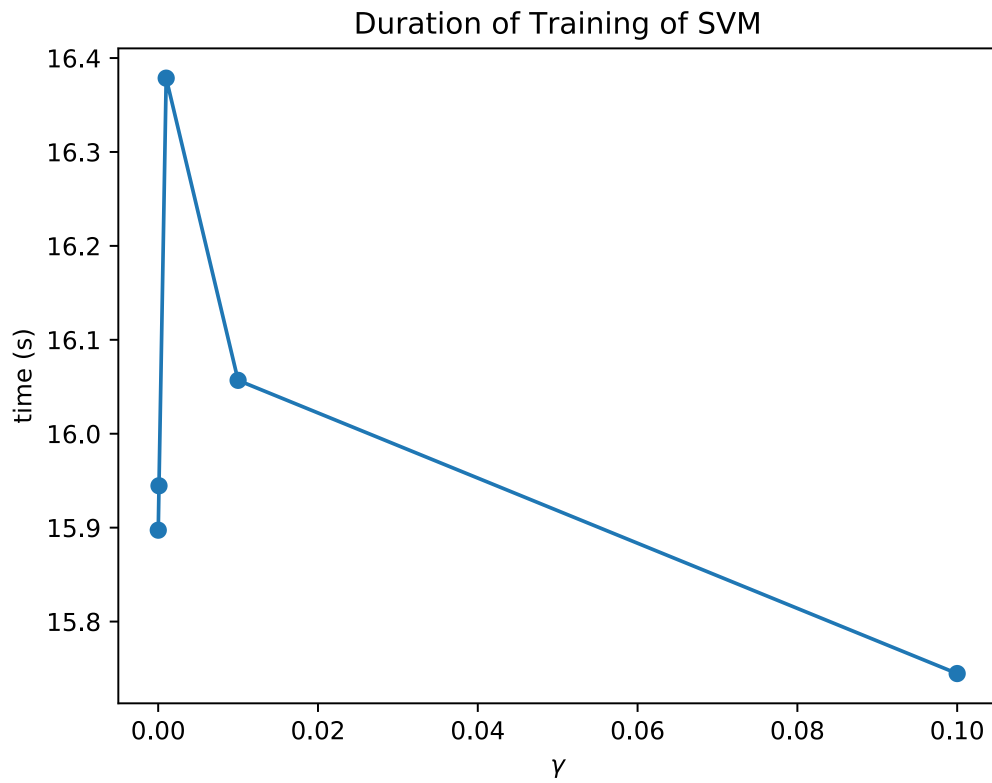


Figure 5.17: Duration of training of the SVM RBF kernel model with maximum iterations = 10^3.

5.4.4 K-Nearest Neighbors

K-Nearest Neighbors differ from the other two algorithms in that it is fast to train but slow to classify, for an offline application it might be suitable as the accuracy is high. There is no significant difference in duration of training time of the algorithm but there is a slight increase of prediction time as the number of neighbors increase. The highest accuracy is achieved for 50 neighbors.

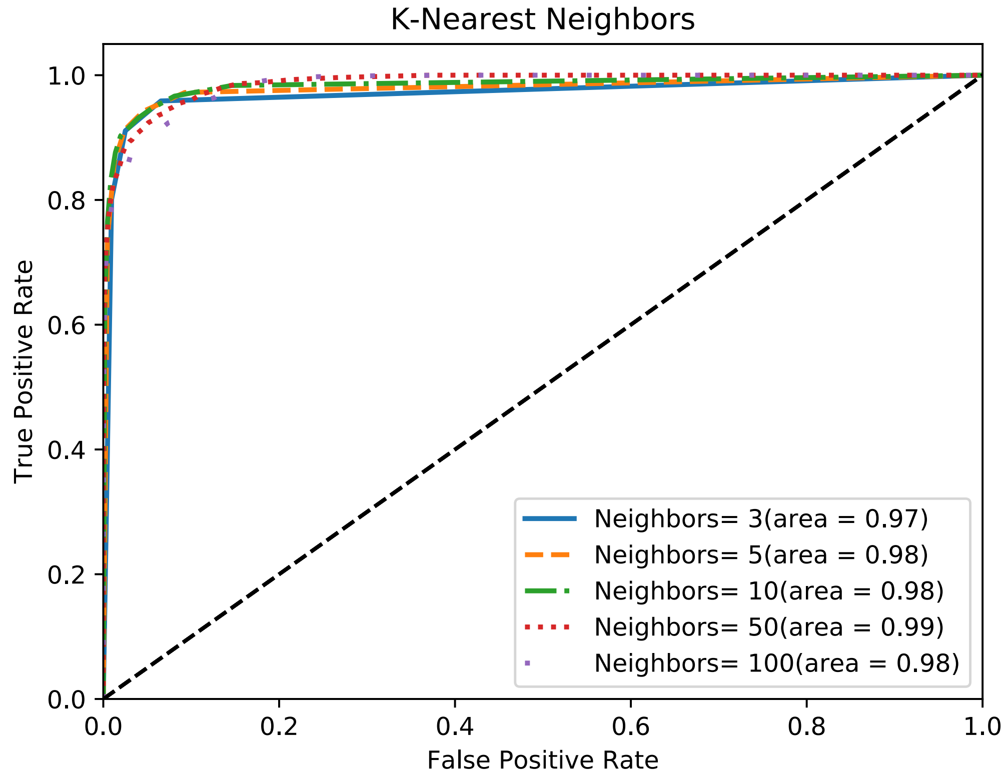


Figure 5.18: ROC curve of the K-Nearest Neighbors classifier, each curve is calculated by using a different number of neighbors.

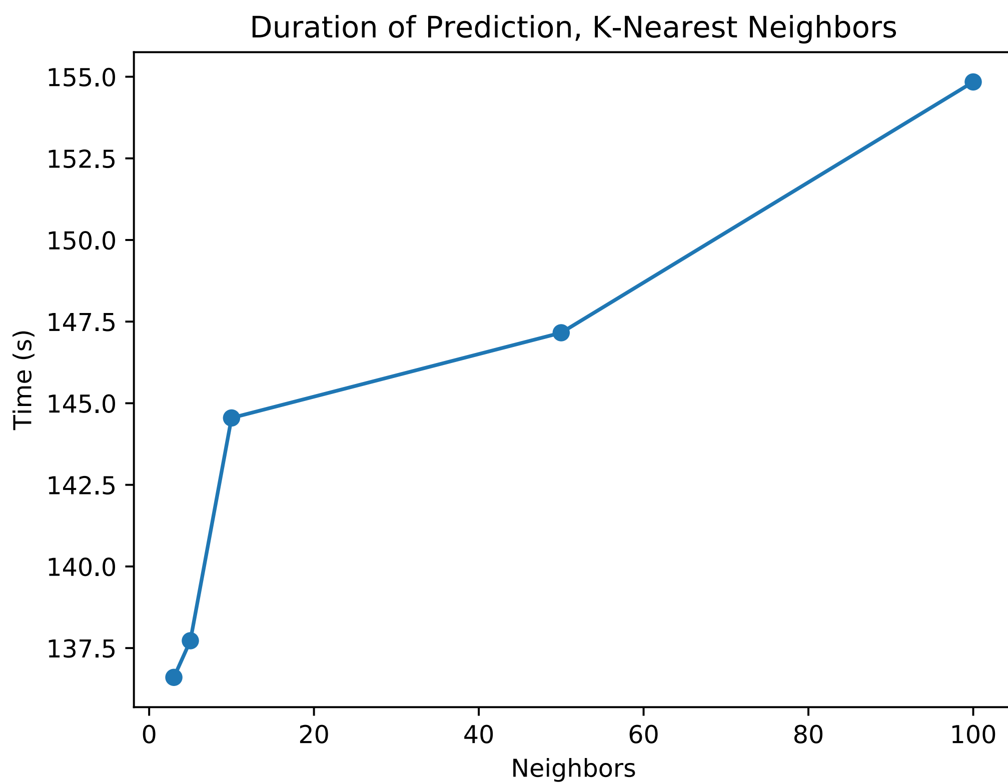


Figure 5.19: Duration of classification using K-Nearest Neighbors. The time for classification increases as the number of neighbors does

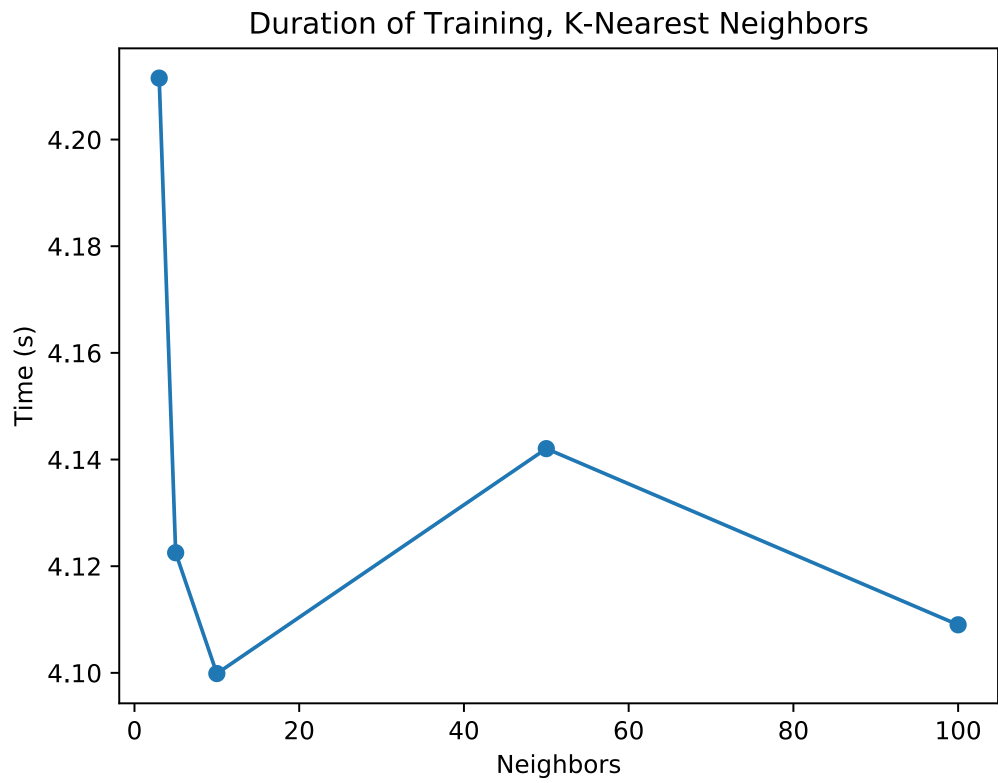


Figure 5.20: Duration of training of the K-Nearest Neighbors models.

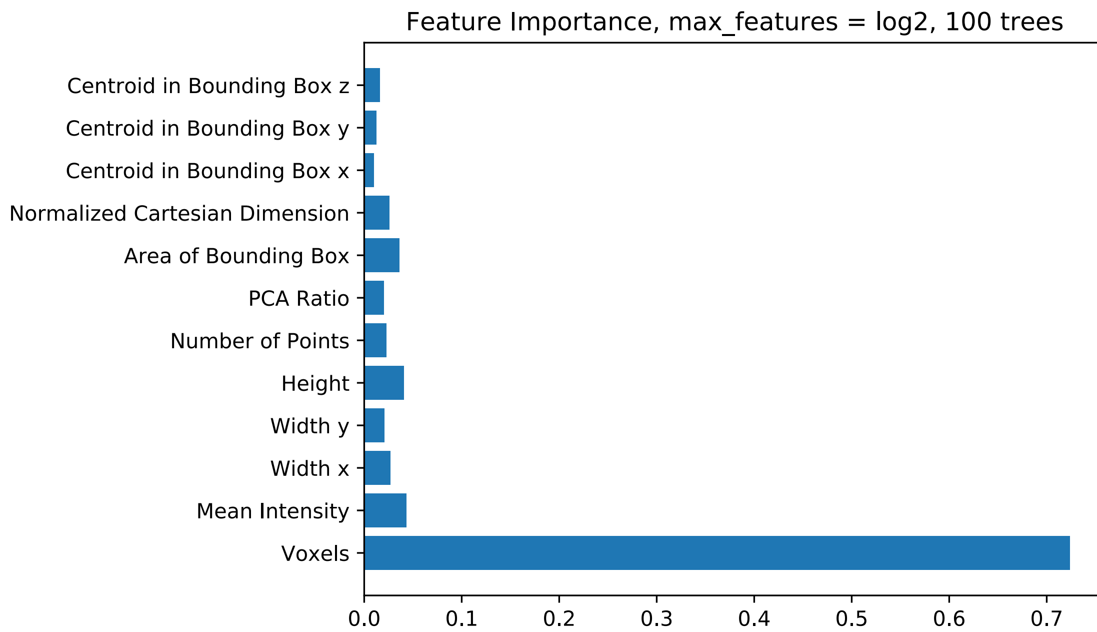


Figure 5.21: The importance of each feature for Random Forest with 100 trees and $\log_2 n$ max features.

5.5 Features

The importance of features for different Random Forest classifiers. The difference in feature importance between different values of max features is much more significant than between different number of trees. The importance of each voxels is combined and is displayed as only one feature making it the most important feature for all classifiers. It is interesting that there is a large difference in its importance between the different values of max feature, varying from around 0.45 to over 0.7.

Figure 5.24 shows the duration of the feature extraction for the features used in this project, these are the average times of extraction for a typical cluster. It should be taken in to consideration that there are usually hundreds or thousands of cluster in each frame depending on the location.

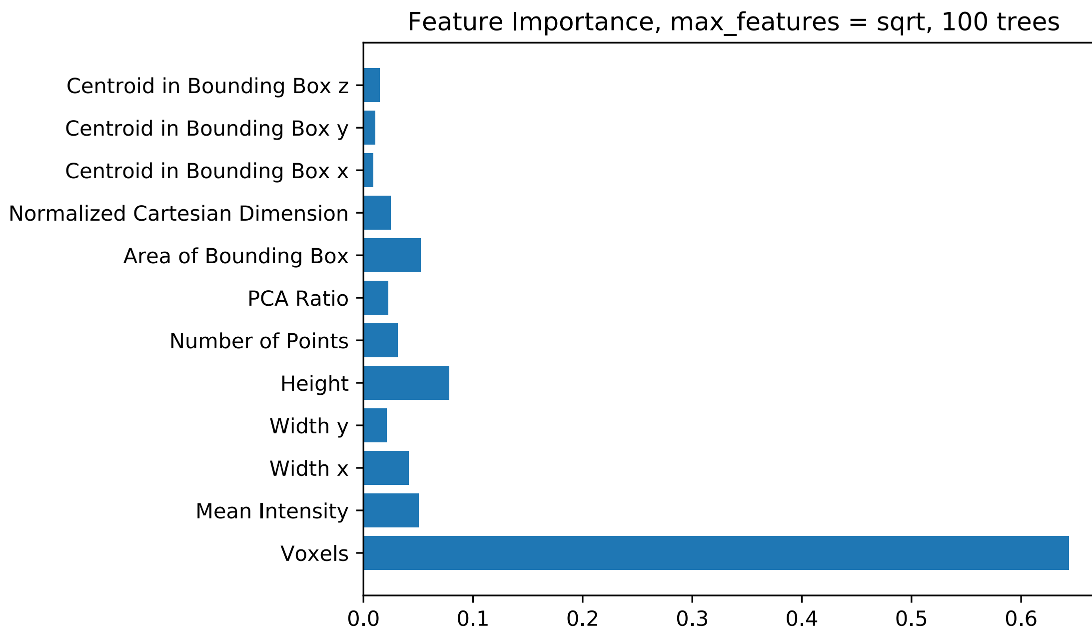


Figure 5.22: The importance of each feature for Random Forest with 100 trees and $\sqrt{n}$ max features.

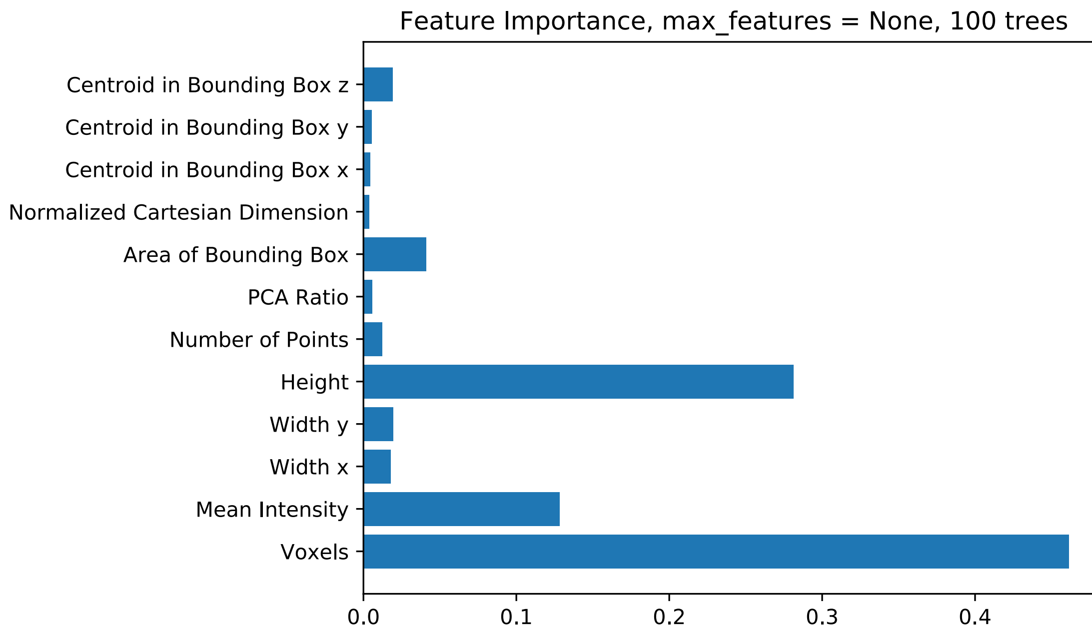


Figure 5.23: The importance of each feature for Random Forest with 100 trees and n max features.

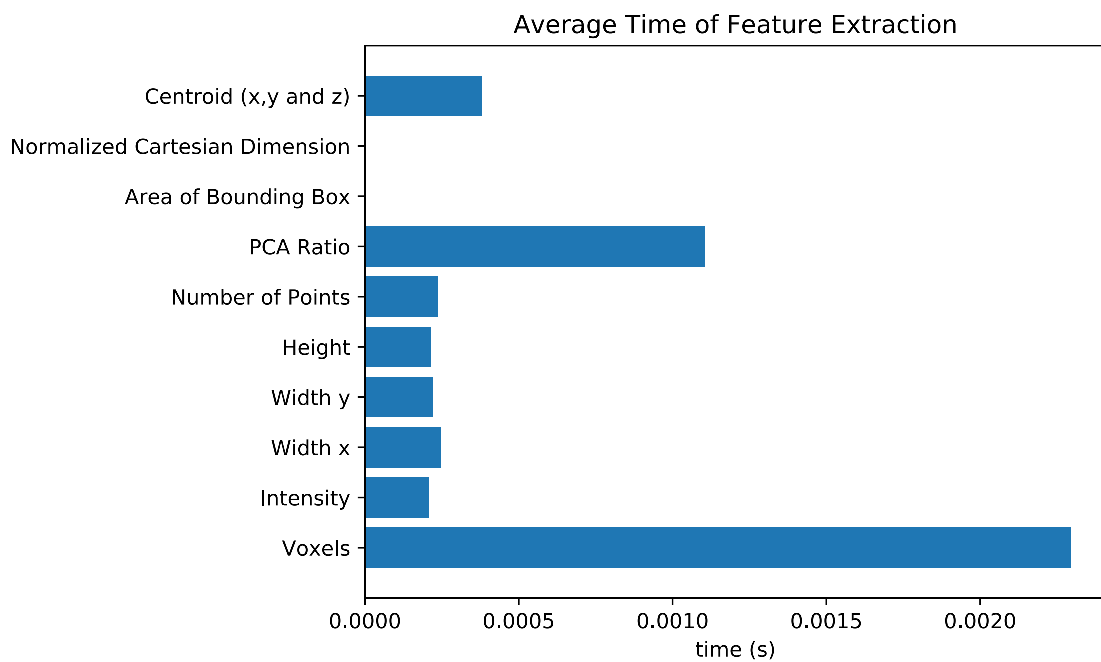


Figure 5.24: The duration of the extraction for all features, note that the x, y and z centroids are extracted simultaneously.

Chapter 6

Discussion

6.1 General Discussion

During this project the goal was to create a pedestrian detector and the focus was therefore on classifying humans. There is support for other classes such as cars or cyclists. The classifiers are only trained on human or non-human data and these are the two only categories. What classes that are needed depend on the application of the product, and a large amount of training data is required to make a working program. It should be noted that both the training and test dataset are limited and do not contain enough data for the algorithm to be successful in all locations and would need to continuously be retrained with more data. However, in general the algorithm is able to detect humans if there are any in the scene, there are exceptions such as people in positions other than standing or walking as the training data mostly contains people walking.

6.2 Classifiers

The most important variables when it comes to classifiers are time complexity and accuracy, even a difference in accuracy of 1% is noticeable when using the algorithm on a scene as a scene usually contains hundreds of point clouds sent to the classifier. The classifier of choice is Random Forest as it is fast at classifying and has a high accuracy. The choice of number of trees is a bit arbitrary as the speed of a prediction (see Figure 5.12)is about the same while the number of trees is less than 100 and they all achieve 99% accuracy. If the training dataset changes a new evaluation of the classifiers should be performed to find the optimal solution for the new dataset. A disproportional amount of clusters falsely being classified as humans are clusters placed in the overlapping area between two channels of the sensor (as seen in Figure 2.2).

The best performing classifier is Random Forest. With a high accuracy and low training

and predicting duration is seems to be the best classifier.

The SVM has a relatively good accuracy of up to 97% (it is however lower than Random Forest). The speed of prediction is slightly slower than Random Forest. It is possible that it performs better on different training datasets and it should be tested if any changes to the dataset is done. Since best performing value of γ is 10^{-4} when the initial test was performed it was investigated if there might be a better performing value of γ as there is a possibility to reach higher accuracy (see Figure 5.15). However, no better accuracy was achieved. As the classification speed does not change much with different values of γ the γ with the best accuracy should be used.

K-Nearest Neighbors has a high accuracy but the classification speed is much too low. Having a prediction time of seconds is not realistic for a real time solution.

Currently the algorithm is able to detect humans that are up to 20m from the sensor, this is most likely because the majority of the training data has been from shorter distances (<20m). Other problems are the distance between points at larger distances and the amount of points in a human cluster. This affects the clustering algorithm inside of the box that is being swept through the scene, making it unable to create clusters (either because of the distance between points being too large or that there are not a sufficient amount of points).

6.3 Features

The combination of all voxels are the most important features for all Random Forest classifiers which is not surprising as it is over 500 features. Most of the voxels are however very insignificant as some voxels are bound to be empty for most clusters. It was rather surprising how much the importance of the height differs between the different classifiers. It was theorized from before that it would be a very important feature but not that it would matter this much. The importance of the width x and y is very low for all classifiers, this is not surprising as all clusters has a maximum width of 0.7 in both the x and y- direction (because of the size of the sweeping box), if the size of the sweeping box was larger, the features would most likely increase in importance because of the inclusion of much larger clusters.

As previously mentioned there is not any requirement of a real time application for this project. However, for better usability a quicker algorithm is to be preferred. The voxel features are the most important features in this implementation but it is also the most time-consuming one. The voxel extraction is proportional to the number of voxels used, an investigation on the optimal number of voxels would have to take both accuracy and duration in to consideration. The PCA Ratio feature is rather time-consuming and of little importance and should probably be discarded in future applications.

Chapter 7
Future Work

7.1 Annotation Tool

The annotator has a lot of room for improvement. A problem with current implementation is that the radius for DBSCAN which can be modified modifies the DBSCAN for the entire scene. It would be preferable to have a finer control over the DBSCAN by setting up regions in the scene. The problem with the current solution is that you normally want a very low radius when close to the LiDAR, something close to 0.02 is normally fine. When far away from the LiDAR much larger values are needed. One wants as low radius as possible to make sure that clusters do not melt together into one cluster. It is probably a good idea to be allowed to change the minimum points needed for DBSCAN too, currently it is set to 20. However, subjectively speaking, it is hard to determine a human shape in clusters with less than 20 points.

Preferably the GUI which uses Matplotlib should be replaced with a better GUI that is made for rendering 3D objects in real time. Apart from Matplotlib seems very slow when plotting many points it is also very hard to control the camera to get a good view of the scene and therefor drastically reduces the user experience. When we were annotating we played the video sequence in an external video program, this could be built into the software so that the operator clearly can see what is happening. Even if the rendering of the scene is perfect as should be expected in a final product the looping of the video sequence still helps a lot because you can clearly see movements which can not be captured in a single frame. Showing the scene from several different angles in multiple windows will probably also enhance the user experience to quickly get a good overview of the scene.

Adding the possibility to go back to previous frames would be a very handy and expected feature to allow more possibilities of smoothly fixing mistakes.

Currently Lucas Kanade optical flow [11] is used to track humans which works fine as long as only humans are tracked, however if there is a need to track several different objects with large differences in speed, e.g. cars,humans and bikes in the same scene a different

tracking algorithm would probably be needed. There is an optical flow algorithm based on machine learning which apparently gives very accurate results for both fast and slow motions that could be considered.

Since our classifier works reasonably well one could seriously consider building it in to the annotator and let it automatically label humans. The operator can then simply confirm the correct labeling and remove the few false positives.

7.2 Classifier

The amount of training data and the variation of it should be increased, since the training data contained exclusively adults walking it could be sensitive to other types of movement, positions, clothes or body shapes and sizes. It is also possible to implement other classes than humans such as animals or vehicles depending on the real life application. To increase the amount of training data to decrease the risk of overfitting, it is usually advised to use jittering on the training data, this was not done in this project. [6] uses jittering such as rotation and size alterations on point clouds. There is currently no review of training data, this could be done by examination of outliers contained in the training dataset.

The features used can be adjusted to increase successful classifications and to decrease time complexity. This is especially important if the aim is to create a real-time implementation. There are infinite options when it comes to feature selection, the ones used in this project are rather basic ones. Other feature engineering such as binning could be used on features that currently is numeric.

Another approach would be to feed a neural network with the voxels before using a classifier.

Further optimisations could possibly be done when designing the model of the classifier. A few parameters were tested during this project to get a direction of where the optimal values were.

A Region Proposal Network (RPN) such as the one used in VoxelNet [6] is believed drastically decrease the run time of the algorithm as there would no longer be any need to sweep a box through the scene, it would however require a neural network to be implemented.

Bibliography

[1] J. Mei, B. Gao, D. Xu, W. Yao, X. Zhao, and H. Zhao, "Semantic segmentation of 3d lidar data in dynamic scene using semi-supervised learning," *CoRR*, vol. abs/1809.00426, 2018.

[2] T. Czerniawski, M. Nahangi, S. Walbridge, and C. Haas, "Automated removal of planar clutter from 3d point clouds for improving industrial object recognition," 07 2016.

[3] K. Kidono, T. Miyasaka, A. Watanabe, T. Naito, and J. Miura, "Pedestrian recognition using high-definition lidar," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, pp. 405–410, June 2011.

[4] C. Premebida, O. Ludwig, and U. Nunes, "Lidar and vision-based pedestrian detection system," *J. Field Robotics*, vol. 26, pp. 696–711, 2009.

[5] T. Lin, D. S. Tan, H. Tang, S. Chien, F. Chang, Y. Chen, W. Cheng, and K. Hua, "Pedestrian detection from lidar data via cooperative deep and hand-crafted features," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, pp. 1922–1926, Oct 2018.

[6] Y. Zhou and O. Tuzel, "Voxelnet: End-to-end learning for point cloud based 3d object detection," *CoRR*, vol. abs/1711.06396, 2017.

[7] M. Herzog and K. Dietmayer, "Training a fast object detector for lidar range images using labeled data from sensors with higher resolution," 05 2019.

[8] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography .," *Communications of the ACM.*, vol. 24, no. 6, pp. 381–395, 1981.

[9] A. Hast, J. Nysjö, and A. Marchetti, "Optimal ransac – towards a repeatable algorithm for finding the optimal set.," *Journal of WSCG*, vol. 21, p. 21–30, 2013.

[10] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise.," *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, p. 226–231, 1996.

[11] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision.," *Proceedings of Imaging Understanding Workshop*, pp. 121–130, 1981.

[12] L. E. Raileanu and K. Stoffel, "Theoretical comparison between the gini index and information gain criteria," *Annals of Mathematics and Artificial Intelligence*, vol. 41, pp. 77–93, May 2004.

[13] T. K. Ho, "Random decision forests," in *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, (Washington, DC, USA), pp. 278–, IEEE Computer Society, 1995.

[14] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, 1995.

[15] T. Cover and P. Hart, "Nearest neighbor pattern classification.," *IEEE Transactions on Information Theory*, vol. 13, pp. 21–27, 1967.

[16] R. J. Samworth, "Optimal weighted nearest neighbour classifiers," *Ann. Statist.*, vol. 40, pp. 2733–2763, 10 2012.

[17] R. Girshick, "Fast r-cnn," in *International Conference on Computer Vision (ICCV)*, 2015.

[18] "Hitachi automotive and industry lab semantic segmentation editor." `https://github.com/Hitachi-Automotive-And-Industry-Lab/semantic-segmentation-editor`. Accessed: 2019-07-23.

Chapter 8
List of Changes

Since 2019/07/22

- Changed scale of some figures
- Rephrased certain vague parts.