Institute for Computer Science VII
Robotics and Telematics

Master's thesis

Semantic Classification in Uncolored 3D Point Clouds using Multiscale Features

Michael Neumann

May 2020

First supervisor: Prof. Dr. Andreas Nüchter
Second supervisor: Prof. Dr. Klaus Schilling

Abstract

The interest in 3D point clouds steadily grows. While the semantic segmentation of 2D images is nowadays a very common and well-researched field, the assignment of semantic labels in 3D is lagging. This is partly attributable to the fact that prelabeled training data is only rarely available since not only the training and application of classification methods but also the manual labeling process are more time-consuming in 3D.

This thesis focuses on the more classical approach of first calculating features and subsequently applying a classification algorithm. Existing handcrafted feature definitions are enhanced by using multiple selected reductions of the point cloud as approximations. They serve as input to train well-studied classifiers like random forests and support vector machines. Both classification methods are compared.

In contrast, deep learning approaches emerged during the last years. Our results are compared to the recently published Kernel Point Convolution method which can be paraphrased as a convolutional neural network for direct 3D input. A comparison shows that there are applications for both modern and classical methods.

Additionally, the developed tool *Blender2Helios* is presented. It enables the smooth conversion of existing 3D scenes to semantically labeled 3D point clouds. We show that this artificial data is well-suited for training real-world classifiers.

Zusammenfassung

Das Interesse an 3D-Punktwolken wächst stetig. Während die semantische Segmentierung von 2D-Bildern heutzutage ein sehr verbreitetes und gut erforschtes Themengebiet ist, hinkt das semantische Labeln in 3D hinterher. Dies ist zum Teil darauf zurückzuführen, dass vorgelabelte Trainingsdaten eher rar verfügbar sind. Grund dafür ist, dass nicht nur das Trainieren und Anwenden von Klassifikationsmethoden, sondern auch der Prozess des manuellen Labelns in 3D zeitaufwendiger ist.

Diese Arbeit konzentriert sich auf den eher klassischen Ansatz, zunächst Features zu berechnen und anschließend einen Klassifikationsalgorithmus anzuwenden. Bestehende handgefertigte Merkmalsdefinitionen werden durch die Verwendung mehrerer ausgewählter Reduktionen einer Punktwolke als Approximationen ergänzt. Sie dienen als Input, um bekannte Klassifikatoren wie Random Forests und Support Vector Machines zu trainieren. Beide Klassifikationsmethoden werden verglichen.

Im Gegensatz dazu sind in den letzten Jahren Deep Learning Ansätze entstanden. Unsere Ergebnisse werden mit der kürzlich publizierten Kernel Point Convolution Methode verglichen, die als Convolutional Neural Network für direkte 3D-Eingaben beschrieben werden kann. Im Vergleich zeigt sich, dass es Anwendungsfälle sowohl für die modernen als auch für die klassischen Verfahren gibt.

Zusätzlich wird das entwickelte Werkzeug *Blender2Helios* vorgestellt. Es ermöglicht die reibungslose Umwandlung bestehender 3D-Szenen in semantisch gelabelte 3D-Punktwolken. Wir zeigen, dass diese künstlichen Daten gut für das Training von Klassifikatoren in der realen Welt geeignet sind.

Acknowledgments

This master's thesis describes my last project as a computer science student at the University of Würzburg. During my research I learned a lot about laser scanning and 3D point clouds including their semantic classification. I was also able to contribute code to interesting software projects like *3DTK*.
This project was only possible with the support of many people.

Special thanks to my adviser, **Prof. Dr. Andreas Nüchter**, who guided me the right way and took the time for discussions every week. The knowledge he transmitted in his lectures – *Advanced Automation*, *3D Point Cloud Processing*, and *Photogrammetric Machine Vision* – within the last years was very helpful. He always had an ear for my questions and allowed me great freedom for my work. Additionally, the employment as teaching assistant supervised by him provided deeper insights in interesting topics like ROS and C++ programming.

Also many thanks to my second supervisor, professor in ordinary **Prof. Dr. Klaus Schilling**, for enabling and grading this work.

I am also grateful for the help of all my other **colleagues at the chair of computer science VII**, particularly **Dr. Dorit Borrmann**, **Dr. Christian Herrmann**, and **Johannes Schauer**, for giving helpful information and providing access to hardware and software resources.

In addition I want to thank my **family and friends**, especially my sister Christina and my parents Birgit and Thomas, for their support while working on this thesis. Your help is deeply appreciated.

Further thanks to:

Prof. Dr. Bernhard Höfle and his team for their great work on the open-source software *Helios* and insightful conversations about it.

Prof. Dr. Thomas P. Kersten for providing a detailed realistic Unreal Engine scene to apply and test *Blender2Helios* on.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Semantic classification in 3D point clouds means the process of assigning semantic labels to each 3D point in a cloud. Some major differences to the same task in 2D exist. Besides the considerably high volume of data, it lacks a fixed grid-structure. This makes it hard to apply common algorithms for 2D images on point clouds. The variable density of the data is another difficulty. It highly depends on the distance to the laser scanner and is influenced by occlusions.

Just a few years ago one typical approach for segmenting point clouds was split into two parts. First, handcrafted local features of each point were calculated. For this, Principal Component Analysis was applied on a number of neighbors. Originally, the size of this neighborhood, respectively the number of neighbors was fixed.[114] In 2014 an improvement with varying neighborhood sizes, called *optimal neighborhood size selection* was proposed. Up until 2018 various other improvements were developed and published. For instance, [45, 104] proposed the calculation of features on many different scales but with a different neighborhood definition. Second, common classification algorithms were applied on the features. While basically any general classifier is applicable, random forests showed high accuracies and became the standard.[115]

With the availability of datasets with billions of labeled 3D points like *Semantic3D* [43] and capable hardware, deep learning emerged in this field of research. Thus, the manual definition of features became redundant. Some early approaches projected 3D points on 2D planes to create artificial images on which existing methods were applied.[15, 64, 102, 103] Meanwhile, a direct processing of 3D data is done most often. The published solutions reach from *superpoint graphs* [63] to *RandLA-Net* [52] and *KPConv* [105]. The latter approach, which is very similar to known convolutional neural networks, is described in more detail in this thesis. As every year new methods are developed, there is currently no de facto standard. A good evidence about methods with currently the highest accuracies is provided by various publicly available benchmark tests like the Semantic3D challenge at `http://www.semantic3d.net/view_results.php`.

This thesis describes the work of updating the classical approach with handcrafted features using current research. We propose the calculation of features on selected point cloud reductions as a high-performance approximation. A random forest and two support vector machines are used for the later classification. Additionally, the output of KPConv, a proponent of the deep learning classifiers, is compared to them.

For representative results, all parameters of the learners are tuned on only one dataset. Their performance is reviewed on a total of four different datasets. This resembles typical applications and differentiates our work from others. The used datasets consists of terrestrial (TLS) and mobile laser scans (MLS) of outdoor scenes.

One set of scans, the *Sim2Real* dataset, allows the examination of classifiers that were only trained with simulated data. For this purpose, we implemented Blender2Helios that allows to create semantically labeled training data of existing 3D scenes.

It has to be noted that our understanding of *classification in point clouds* is sometimes also called *segmentation of point clouds*. Both mean the same. In contrast, some authors use the term *classification of point clouds* to denote the task of assigning one label to a whole point cloud.

This thesis is best viewed in color as it shows multiple point clouds where semantic labels are indicated by different colors.

1.1 Outline

This thesis is structured as follows.

Chapter 2 contains preliminaries for understanding this work. It transfers basic knowledge about laser scanning and 3D point clouds as well as common data structures for efficiently storing and querying them. Additionally, the Principal Component Analysis that is later used in the calculation of features is introduced. The last section describes the main concepts of the used classifiers: Random forests, support vector machines, and neural networks.

The rest of this thesis is organized in two parts. Part I on "Tools and Datasets" is split into three chapters.

In chapter 3 the variety of tools that was used during this research is introduced. This includes a detailed introduction to the 3D point cloud processing toolkit *3DTK - The 3D Toolkit* and its internal functioning. Furthermore, the machine learning software *Weka* and its typical *.arff* file format is presented. In addition, the 3D suite *Blender* and the LiDAR operations simulator *Helios* for which we implemented an interface are described.

Four different datasets are presented in chapter 4. Besides the small *Oakland* and *Paris-Rue-Madame* MLS datasets, an overview of the much bigger *Semantic3D* TLS dataset is given. The point clouds created during the work on this thesis, which we call *Sim2Real*, are also introduced.

Chapter 5 on *Blender2Helios* provides information about the developed interface. It describes the *Blender* add-on that allows the conversion of existing scenes to *Helios* compatible *.xml* and *.obj* files for laser scan simulations. Aside from the occurred challenges during implementation, its usage and pseudocode are shown.

Semantic classification is examined in the second part.

The concept, implementation, and tuning of different classifiers on the *Semantic3D* dataset is depicted in the sixth chapter. For the more classical approach the search for an optimal neighborhood size is discussed. Furthermore, our understanding of local features on multiple scales as an approximation of bigger neighborhoods in the original data is presented. At the end, the process of parameter tuning is described for a random forest, two support vector machines, and one neural network approach.

All classifiers were trained and tested on the different datasets in chapter 7. A comparison of their results, in terms of accuracies and example images, is shown. While mainly focusing on the classification decisions, the run time of the different methods is also taken into account.

Conclusions about the applicability of the considered methods are drawn in the last chapter. Additionally, thoughts about future work are provided.

The appendix contains pseudocode of helper functions of *Blender2Helios* as well as more detailed results of the process of parameter tuning and the classifier comparison. References to it are given in the text.

1.2 Scientific Contribution

The research done for this thesis contributed to different scientific questions and tasks described in the following paragraphs.

Proposal of Selective Multiscale Feature Calculation In chapter 6 we propose the calculation of features on multiple but selected scales. In comparison to other work like [45], we exploit an optimal neighborhood definition introduced in [115]. Additionally, the set of used features reduces the assumptions that were made, like a consistent height of the laser scanner. The whole code is open source and released with *3DTK - The 3D Toolkit*. Calculated features are stored in a format that easily allows further processing with machine learning software.

Comparison of Classical and Deep Learning Approaches A comparison of the extended classical approach and *KPConv* as a representative of the deep learning approaches is done in chapter 7. Besides the common setup of using random forests with handcrafted features, we also investigate the applicability of support vector machines for this task. Previous comparisons like the results in the *Semantic3D* challenge often lack a common basis as some algorithms use more data, e.g. color information, than others. Furthermore, since in the last years mainly deep learning methods were promoted, these comparisons often miss actual input of classical methods.

Implementation of Semantic Labels in *3DTK* During the work on this thesis some enhancements for the open source software *3DTK* were implemented. These include two new file formats that contain, besides three-dimensional coordinates, semantic class labels. The graphical user interface of the contained 3D viewer *show* was expanded by a legend that varies depending on the number of classes.

Enabling Highly Detailed and Labeled Laser Scans of Existing 3D Scenes With *Blender2Helios* an interface between the 3D suite *Blender* and the scientific LiDAR operations simulator *Helios* was created. As our results in chapter 7.3 show, training a classifier on artificial data also allows high accuracies on real point clouds. We want to encourage everybody who has access to highly detailed 3D scenes to generate semantically labeled point clouds. Training classifiers well is only possible with a big amount of high-quality prelabeled data. Due to the high expenditure, manual annotation is no alternative.

Chapter 2

Preliminaries

2.1 Laser Scanning and Point Clouds

2.1.1 Methods for Measuring Distances with Light

Laser scanners make use of *time of flight* (ToF) measurements to acquire the ranges to objects in various directions. The measurement range of a laser scanner is typically between a few meters up to some hundred meters. While the whole system is slowly rotating, a tilting mirror allows for fast and exact vertical alignment of the laser beam as shown in figure 2.1. Reflections from the surface of objects are detected by a build-in receiver. Each single measurement can be seen as the polar coordinates defined by the alignment of the laser beam at the moment of this measurement and its calculated distance. A conversion to Cartesian coordinates is done afterwards. The resulting data is a 3D point cloud with many infinitesimal small and unordered points sampled from the real environment. Depending on the actual application, some specifics have to be taken into account. We will discuss some of them in section 2.1.3.[48]

ToF exploits the time light needs to travel. The velocity of light in vacuum, commonly denoted c and shown in equation 2.1, is constant.

$$c = 2.99792458 \cdot 10^8 \frac{m}{s} \tag{2.1}$$

There are three commonly used methods for ToF [48]:

Direct Time Measurement The laser scanner sends out a light impulse and receives the reflected signal after time t. The distance d to the object reflecting the signal is then calculated by $d = \frac{c \cdot t}{2}$. The factor $1/2$ comes from the fact that the distance between the scanner and the scanned object needs to be covered twice. Precise time measurement is the crucial part of this method. For instance, light needs only approximately 1 ns to travel a distance of 30 cm. Specialized ICs like the Acam TDC-GP2 can measure time in the scope of 3.5 ns to 1.8 μs with a resolution of 65 ps. This corresponds to measurements of 52.5 cm to 270 m with an accuracy of about 1 cm.

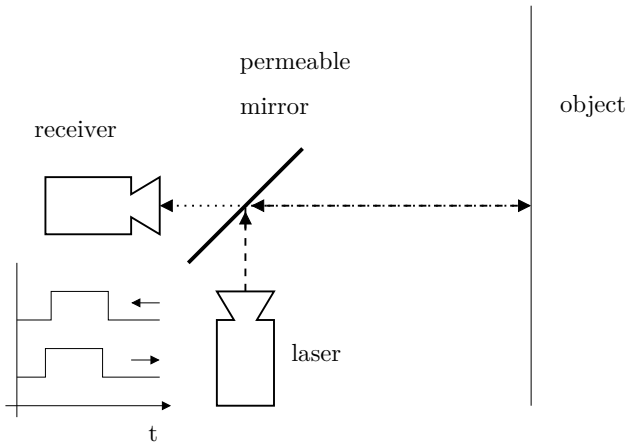
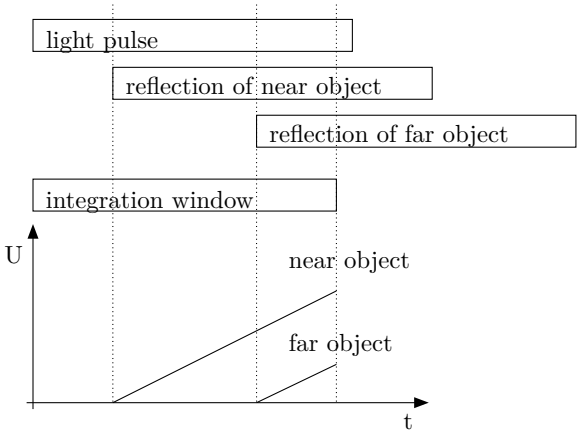
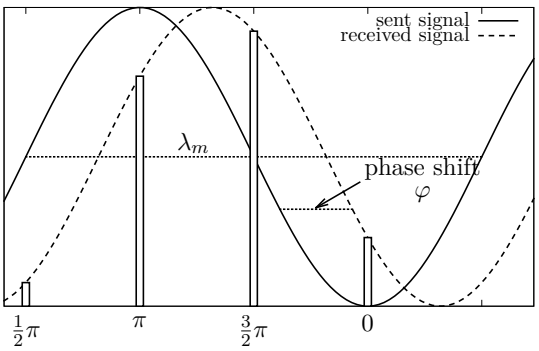


Figure 2.1: Layout of a laser scanner. Adapted from [11].

Pulse Integration The pulse integration method exploits image based technology. Similar to the first method, the scanner sends out a light impulse. At this point an image sensor starts to integrate the luminance for a fixed period of time. For nearby objects the reflected light signal will affect the integration stronger as it impacts the brightness values sooner/for a longer period of time within the integration window. The output, in form of a voltage, is increased. Figure 2.2a illustrates this. In practical applications more than one measurement is done to account for tolerances and ambient light. A first image is taken without emitting light to have a basis of brightness available. These values are later subtracted from the measurements taken with emitted light pulses to eliminate all other light sources than the own emitter.



(a) Principle of pulse integration. Example for two objects with different distances from the scanner.



(b) Illustration of the phase shift between sent and received signal.

Figure 2.2: Methods for *time of flight* measurements. Adapted from [11].

Modulated Light Instead of sending single pulses (discontinuous methods), modulated light allows for continuous measurements. The setup is very similar to the direct time measurement. Its difference is that instead of measuring the time for reflections to occur, the phase shift of a continuously emitted sinusoidal signal is identified. Figure 2.2b shows a very basic implementation of a receiver sampling four times during a full period (0, $\frac{1}{2}\pi$, π, and $\frac{3}{2}\pi$) resulting in four assessed amplitudes a_0, a_1, a_2, and a_3. By means of discrete Fourier transform the phase shift φ is calculated according to equation 2.2.

$$\varphi = \arctan(\frac{a_0 - a_2}{a_1 - a_3}) \tag{2.2}$$

Given the modulation frequency f and therefore the time of oscillation, one calculates the distance d to a reflecting object using equation 2.3.

$$d = \frac{c \cdot \varphi}{4\pi \cdot f} \tag{2.3}$$

Due to the fact that the measured phase shift φ stays between 0 and 2π, the maximum measurable distance is limited. Equation 2.4 shows the calculation of the maximum distance d_{max} where ambiguities start to occur. $\lambda = \frac{c}{f}$ denotes the wavelength of the used laser.

$$d_{max} = \frac{c}{2f} = \frac{\lambda}{2} \tag{2.4}$$

Without this additional constraint there is an infinite number, practically only limited by physical influences, of potentially observed distances as shown in equation 2.5.

$$d_{unconstrained} = \frac{c \cdot \varphi}{4\pi \cdot f} + k \cdot \frac{c}{2f}, \qquad k = 0, 1, 2, 3, ... \tag{2.5}$$

With typical sensors a resolution of the phase shift of 1° is achieved. This results in inaccuracies in the scope of centimeters given a typical modulation frequency of 20 MHz. Increasing the frequency offers a higher depth resolution while decreasing the maximum measurement range.

To overcome the range limitation introduced by ambiguities, two different modulation frequencies (f_1, f_2) are used. While the first phase φ_1 is unambiguous over the entire measurement range providing only a coarse estimation, the second phase φ_2 allows for a much finer resolution within the previously determined and for f_2 unambiguous range window.

In modern laser scanning the first and the last approach are used. Due to the discontinuous measurement in direct time systems and therefore a high ratio of time waiting for reflected echoes, the effective data rate of these setups is smaller than when using continuous systems. In some practical applications, like the Riegl VZ-400i laser scanner, it is possible to have two pulses in the air simultaneously for an increased data rate.[11] Furthermore, a big advantage of the direct time measurement is its long range, often up to several kilometers. Another possible feature of direct time systems is their ability to perform full wave analysis. In the next section this aspect is described in more detail.

2.1.2 Point Clouds

This section focuses on the math we use throughout this thesis to denote point clouds. Other aspects are discussed in more detail in the corresponding chapters about laser scanning.

A point cloud P is an unordered set of $n \in \mathbb{N}$ infinitesimal small 3D points. In contrast to 2D images they are usually not aligned in a fixed grid structure. A single point $\boldsymbol{p_i} \in P$ is given by the column vector $\boldsymbol{p_i} = [p_{i,1}\ \ p_{i,2}\ \ p_{i,3}]^\intercal$. We limit a point to its three coordinates. In other applications more attributes like color information or a reflectance value might be present. Equation 2.6 shows our resulting definition of a point cloud.

$$P = \{\boldsymbol{p_1}, \boldsymbol{p_2}, \dots, \boldsymbol{p_n}\} = \left\{ \begin{bmatrix} p_{1,1} \\ p_{1,2} \\ p_{1,3} \end{bmatrix}, \begin{bmatrix} p_{2,1} \\ p_{2,2} \\ p_{2,3} \end{bmatrix}, \dots, \begin{bmatrix} p_{n,1} \\ p_{n,2} \\ p_{n,3} \end{bmatrix} \right\} \tag{2.6}$$

Given a set of classes $C = \{1, 2, \dots\}$ our later classification task can also be described as finding a correct mapping $c_P : P \to C$.

2.1.3 Specifics and Advanced Features of Laser Scanning

Beam Divergence and Full Wave Analysis

A laser beam is not an infinitely thin line, but a cone. This so-called laser beam divergence results in the possibility of hitting multiple objects during one single measurement. [35] provides a good illustration of this, shown here in figure 2.3.

Typical divergence values are smaller than 1 mrad. For instance a Riegl VZ-400 laser scanning device has a divergence of 0.3 mrad which results in an increase of 30 mm of beam diameter per 100 m distance.

The partial reflection of the laser beam results in multiple echoes. Echoes of nearer objects are received earlier. Their amplitude depends on the share of the laser beam hitting the object. In figure 2.3 three echoes are returned by different branches of a tree. The last echo comes from the house.

There are different ways to handle multiple echoes. Usually only one echo, the first, is used for further processing. Advanced scanners, however, also support *full wave analysis* where more than one range per measurement is returned. This is, inter alia, used in airborne laser scanning where the laser beam is much wider due to the altitude of an airplane. For instance, vegetation can be removed from scans to reveal the terrain surface. This is done by only keeping the last echo, which is returned by ground that was hit between trees or other vegetation. The difference of the first and the last echo allows to infer the height of vegetation within the laser beam cone. [2, 93] include more details and discussions about related cases of use.

Density

In contrast to 2D images which have a uniform density given by the alignment of pixels along a fixed grid, point clouds generated by a laser scanner vary in density. The high variation is introduced by the different distances to objects.

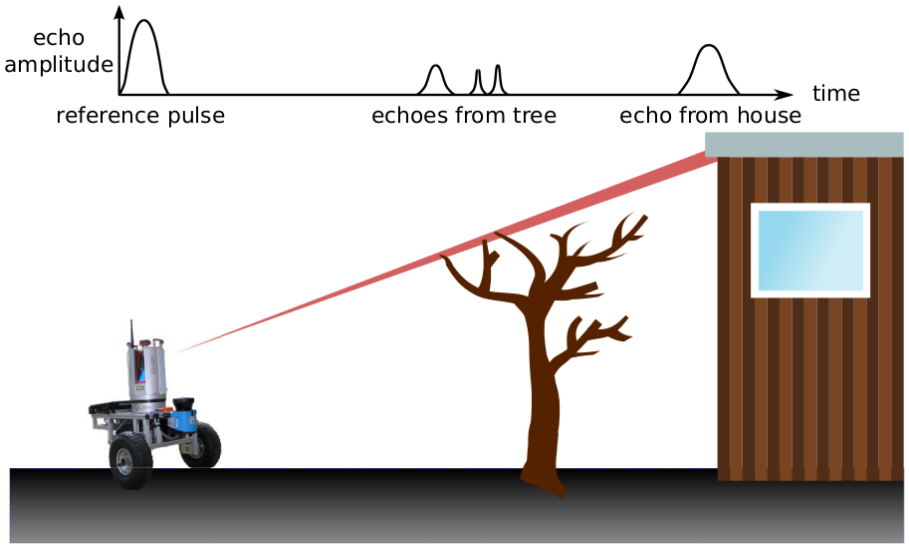


Figure 2.3: Sketch of full wave analysis. Multiple echoes are received by the laser scanner sending out one laser beam. Three echoes were reflected by the tree while the last echo comes from the house. Taken from [35].

As a simple example one can think of a sphere with radius r_1 around the scanner. This sphere with a surface area of A_1 is sampled with the exact same amount of measurements as a bigger sphere with radius $r_2 = 2r_1$. However, the surface area A_2 of the bigger sphere is four times bigger as shown in equation 2.7. In general, the scan density behaves indirectly proportional to the square of the range between the object and the laser scanner. Along the horizontal and the vertical, the point spacing is direct proportional to the reflecting item's distance.

$$A_2 = 4\pi r_2^2 = 4\pi(2r_1)^2 = 4\pi 4r_1^2 = 4A_1 \tag{2.7}$$

In the later presented datasets this behavior led to heavily imbalanced point clouds. Laser scanners are typically mounted on top of some kind of stand or, for instance, on a vehicle. For both scenarios a stable surface is chosen. This results in man-made ground, like streets, being very close to the scanner and therefore sampled with high density.

Occlusions and Scan Registration

Occlusions occur when one object overlays other ones. Everybody intuitively knows this from normal photographs. In laser scanning this effect has a high impact. Depending on the position of the laser scanner, some objects might be partly or completely hidden behind other items. This makes scenes that were converted to point clouds by only one laser scan incomplete. Only the surface pointing towards the scanner is captured. A cube may look like a flat rectangle and a globe like a hemisphere. There is no information about the space behind the surface that reflected the laser beam. More points in a cloud are missing due to the fact that laser scanners usually do not scan 360° vertically. A portion underneath and over the scanner is omitted.

To minimize occlusions, multiple scans are often brought together which is called *scan registration*. By varying the position of the laser scanner, the laser hits objects from different angles and with/without occlusions. When the exact translation and rotation of the scanner between the scans is known, the measurements can be easily merged into one point cloud. As this is usually not the case, the scans need to be matched. A very popular algorithm for solving this problem is the ICP (iterative closest point) algorithm described in [9]. First, different point clouds are aligned by an initial guess, for instance given through GPS data on a vehicle. Then, closest points between different scans are iteratively detected and a translation/rotation is applied to bring those point pairs closer together. With every iteration an error function $E(\boldsymbol{R}, \boldsymbol{t})$ is minimized by changing the rotation $\boldsymbol{R}$ and the translation $\boldsymbol{t}$ between the scans. Equation 2.8 shows a simple version of this error function for two point clouds where $\boldsymbol{p_i} \in \boldsymbol{P_1}$ and $\boldsymbol{q_i} \in \boldsymbol{P_2}$ denote one of n closest point pairs between both scans. Due to the previously discussed occlusions and therefore different sampled points on objects, the error will never become zero. However, after execution the scans are well aligned to each other. Regions that were sampled from more scan positions are denser and surfaces that are at least scanned once in any scan now occur in the resulting point cloud.

$$E(\boldsymbol{R}, \boldsymbol{t}) = \frac{1}{n} \sum_{i=1}^{n} ||\boldsymbol{p_i} - (\boldsymbol{R}\boldsymbol{q_i} + \boldsymbol{t})||^2 \tag{2.8}$$

When performing the classification task on point clouds, training and testing data should contain similar generated point clouds. When comparing samples of a car, a single scan only sees the front, one side, or the back. Depending on the height of the scanner the roof may be visible or hidden. Matching multiple scans will result in the car being sampled from multiple/all sides. This will show a more convex object with points on all sides. Training on single scans and applying the classifier on matched scans and vice versa might drastically decrease classification performance. The goal is to choose homogeneous data.

2.1.4 Used Hardware: Riegl VZ-400

During the work for this thesis laser scans of different scenes were taken. The results of a classifier trained on only simulated data but applied on our real world data of a parking lot at the University of Würzburg are presented later in this thesis. Figure 2.4 shows an image taken during a laser survey.

The Riegl VZ-400 we used for this is a high speed 3D terrestrial laser scanner. It supports both single and multiple echo measurements. Good connectivity is provided by various interfaces, i.e. LAN, WLAN, USB, an integrated L1 GPS receiver, and a camera mount to allow the creation of colored 3D point clouds.

The laser scanner achieves 5 mm accuracy with 3 mm repeatability and has a maximum measurement range of 600 m with the eye safe near infrared class 1 laser. Up to 120,000 measurements can be processed per second. Its vertical field of view is 100°

Figure 2.4: Image of our Riegl VZ-400 in the parking lot in Würzburg.

while it covers 360° horizontally. The angular step width between two consecutive laser shots is configurable down to 0.0024°.[91]

For our scans we chose an angular resolution of 0.04° both horizontally and vertically. This results in a point spacing of 7 cm at a range of 100 m. A high speed scan, which allows a maximum measurement distance of 350 m for highly reflective objects, takes around three minutes with these settings. Such scans contain up to $\frac{100°}{0.04°} \cdot \frac{360°}{0.04°} = 22.5$ million points. As in pratical applications there are often no objects in the sky, typically around 50% of these points are present.

2.2 Data Structures for Storing 3D Point Clouds

As shown in the last section, our laser scans typically contain over ten million 3D points captured within a few minutes. Other hardware like the Velodyne HDL-64E is even capable of executing multiple million range measurements per second.[67] Performing typical operations like the search for the nearest neighbor of one 3D point in these enormous clouds consumes much time when all points are stored in an ordinary list. Due to the three dimensions it is also not possible to just order the points in the cloud for a reasonably better performance.

More advanced data structures like octrees and k-d trees are commonly used to accelerate similar queries. Not only their runtime complexity but also their memory complexity has to be taken into account because of the typical size of 3D point clouds.

2.2.1 Octrees

Octrees are a generalization of binary trees and quadtrees, but for three-dimensional data. In this section we describe their general structure and the implementation used in [36].

Each node of an octree represents a rectangular cuboid in 3D space containing eight child nodes. Each of the children represents one eighth of the 3D space of the parent. Figure 2.5 illustrates this structure down to a depth of two. Defining the maximal depth of the tree or a minimal number of points per node stops the tree from splitting too often. Without stopping rules the tree would grow until only one point per leaf is left, resulting in much overhead. Nodes describing space without any data points are treated as *empty* which means that they are not split anymore. Leaves of the tree contain a list of all 3D points within the volume described by the cuboid.

Memory Efficiency In the software 3DTK, which is introduced later in section 3.1, octrees are implemented in a very memory efficient way with the capability to store over 1 billion points in 8 GB of memory. The creation of an octree in a recursively quicksort fashion can be done in $\mathcal{O}(n \log n)$, where n again denotes the number of points to store. Also, the runtime of typical access, add, and delete operations is in $\mathcal{O}(\log n)$. Figure 2.6 illustrates one node of an octree. The largest portion of each node is used by the child pointer which is 6 bytes in size on 64-bit systems when not using far pointers.[61] It refers to the first child node. All other children are arranged linearly after this first child in memory. Figure 2.7 shows this in more detail. Two additional bytes are reserved for information about the children. The *valid* byte encodes the information if a specific child is part of the list of children. Thus, not every node needs to have

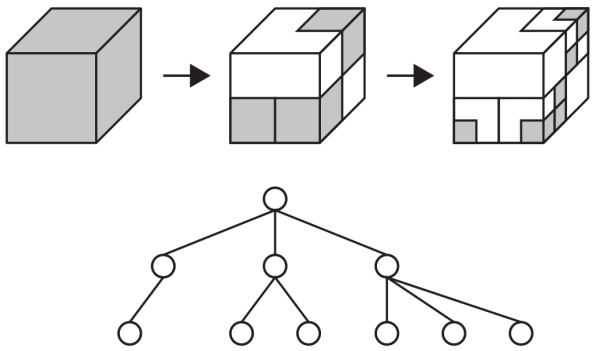


Figure 2.5: Structure of an octree with a fixed maximal depth of two. Empty voxels, colored in white, are not split anymore. The gray leaf voxels in the most detailed illustration of the cube link to a list of points within this volume. Taken from [36].

eight children but empty voxels can be skipped. The fact whether a child is a leaf containing a list of 3D points or a parent node for other children is embodied in the eight *leaf* bits. As only valid children can be leaves, some bit combinations in the valid/leaf bytes can never occur. Therefore a compression to 13 instead of 16 bit is possible. Due to the small saving and concerns about the runtime efficiency this compression is not done. For octrees with constant depth the leaf byte is obsolete even resulting in only 7 bytes of memory per node.[36]

As previously described, each leaf holds a pointer to an array containing all 3D points that lie within its defined 3D volume. The first element encodes the number of points in this voxel. Clearly, the required memory for a whole leaf – including the 3D points – depends on the number of points and the amount and accuracy of attributes that are stored per point. Despite the three coordinates, additional information like color or reflectance values may be part of the array. Further details are skipped here. Interested readers may consult the original paper [36] for details.

Octree based reduction of 3D point clouds

3DTK implements different reduction algorithms for 3D point clouds. Based on the storage of data in octrees it seems reasonable to prune the tree at a predefined depth. In 3DTK this is typically done by providing a threshold as the minimal side length for the cuboids. Only a small

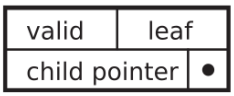


Figure 2.6: 3DTK's memory efficient encoding of an octree node. The *child pointer* is 48 bits on 64-bit systems. *Valid* and *leaf* indicators are 8 bits each. A node therefore only contains 8 bytes of memory. Taken from [36].

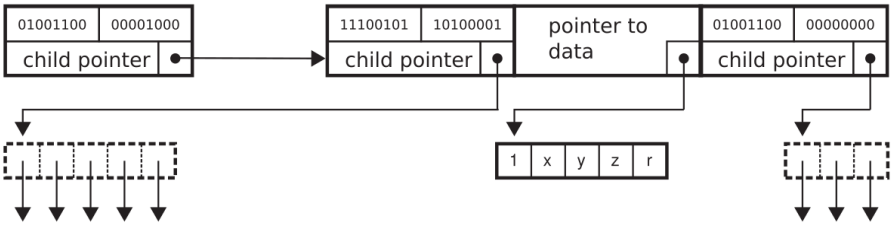


Figure 2.7: Illustration of an octree. The node in the upper left has three children, indicated by the three active *valid* bits. The one bit in the *leaf* section indicates that one of them, the second one, is a leaf node. Instead of storing eight child nodes, empty nodes are suppressed in this implementation resulting in a smaller memory footprint. The leaf node contains a pointer to an array, given by the number of points, here 1, and the points themselves. Taken from [36].

number of up to i 3D points per every new leaf is kept. This results in an upper boundary for the point density. Sparse regions are not affected as the number of points per node is already smaller or equal to i. A special case is $i = 1$ where each leaf only keeps one single 3D point. This point is either picked randomly from the subordinated points or the center of the cuboid is used as a representation. In our work we only use the reduction with $i = 1$ and sample this point randomly.

2.2.2 *K*-D Trees

K-d trees, where k denotes the dimension of the search space, were first introduced in [7]. Like octrees, k-d trees split multidimensional space and enable for efficient queries. However, discrimination is done based on only one attribute at a time resulting in a binary search tree. Regions without any points are also exploited for optimization.

Figure 2.8 illustrates the basic implementation of an optimal k-d tree for a number of n two-dimensional points. Each point $\boldsymbol{p_i}$ is therefore given by $\boldsymbol{p_i} = [p_{i,1}\ \ p_{i,2}]^\intercal$ for $1 \leq i \leq n$. On the root and all even levels, splitting is done based on the values of $p_{i,1}$. In contrast, on all other levels, split decisions are made based on $p_{i,2}$. Each node in the search tree contains one

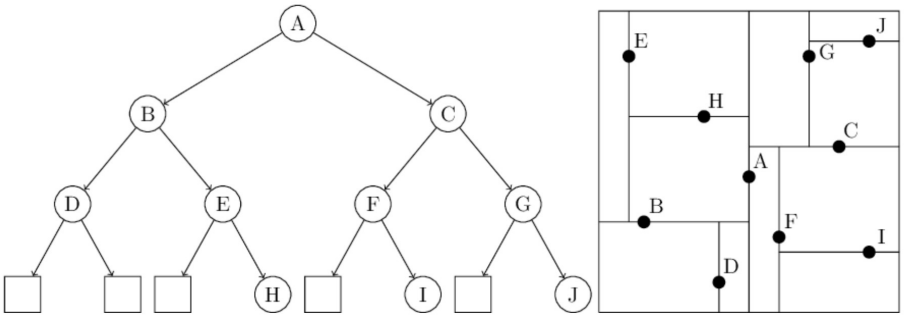


Figure 2.8: Example of an optimal k-d tree. On each level the median of the current split axis was used to divide the data. Taken from [11].

point $\boldsymbol{p_i}$, whereas all *smaller* points can be found in the left subtree and all *larger* points in the right subtree. Smaller and larger here means a lower or higher value respectively in just the corresponding split attribute. One can think of a hyperplane, perpendicular to the split axis, dividing the space into two subspaces at each node. An optimal k-d tree is a balanced one, i.e. the heights of both subtrees differ by at most one and they are recursively balanced. So each node can be reached from the root by a path of length $\lfloor \log_2 n \rfloor$ at most. To create an optimal k-d tree, each split is done at the median point with respect to the split attribute on this level and only the remaining points. In the example shown in figure 2.8, the first split was performed at point A which is the median when projecting all points on the horizontal axis. Next, B and C are the medians in the resulting left and right subtree, respectively, splitting the remaining points by their second attribute value. It depends on the application if spending much time for creating an optimal k-d tree is favored over creating a tree quickly and tolerating higher run times for search queries. A significant speedup can also be achieved by storing the bounding volume of each subtree. Empty spaces can therefore be excluded in search queries. In the basic implementation each node limits the space in just one direction. In 2D space four of these constraints are needed to fully define a bounding box.[11]

The implementation of k-d trees from 3DTK is used throughout this thesis. It has shown to perform well for 3D points and nearest neighbor search. In this implementation all points are stored within leaf nodes. Intermediary tree nodes are used to store the split axis, pointers to child nodes, and the center, the side lengths, and the radius of the bounding box surrounding the contained points. During the creation, which is done recursively, the axis with the largest extent is chosen as the split axis. If the region of one recursion step consists of only a maximum of 10 points, a leaf is created. The up to 10 points are stored in a list. This results in much less overhead for the data structure in comparison to splitting until only one point is left per leaf node. Algorithm 1 shows the creation of a k-d tree in pseudocode.[11]

2.3 Principal Component Analysis

Principal Component Analysis (PCA) forms the basis for multivariate data analysis. It is used in face recognition, image compression, and is a common technique for finding general patterns in high-dimensional data.[100] Its first occurrence was in [82] back in 1901 in biological context. Later, much work was done in the field of probability theory [49, 55, 69] and image processing [78, 108].[41]

PCA is often used to create an approximation of high-dimensional data while keeping the introduced error minimal. To achieve this, a transformation into a vector subspace is performed. The eigenspace of the data's covariance matrix is suited for this, which is spanned by the eigenvectors $\boldsymbol{e_i}$. The corresponding eigenvalues λ_i can be used to determine the variance of the data in the directions of the new vector space. The projection with the greatest variance builds the first/main axis, called first principal component. Smaller variances build other principal components in descending order. So cutting off eigenvectors with small eigenvalues minimizes the quadratic error while simplifying the data.[41]

Algorithm 1 3DTK's implementation to create a k-d tree. Adapted from [11].

Input: Point cloud P
Output: K-d tree with all points of P

```
1: function KDTREE(P)
2:     if |P| ≤ 10 then        // Create leaf node
3:         this.points ← P
4:         this.nrPts ← |P|
5:         return this
6:     else        // Inner node
7:         this.nrPts ← 0
8:         Determine parameters this.center, .d1, .d2, .d3, .r^2 of bounding box
9:         this.splitAxis ← argmax_{i∈{1,2,3}}(this.d_i)
10:        this.splitValue ← center_splitAxis
11:        left ← empty point cloud
12:        right ← empty point cloud
13:        for all p_i ∈ P do
14:            if p_{i,splitAxis} < this.splitValue then
15:                Add p_i to left
16:            else
17:                Add p_i to right
18:            end if
19:        end for
20:        this.childLeft ← KDTREE(left)
21:        this.childRight ← KDTREE(right)
22:        return this
23:    end if
24: end function
```

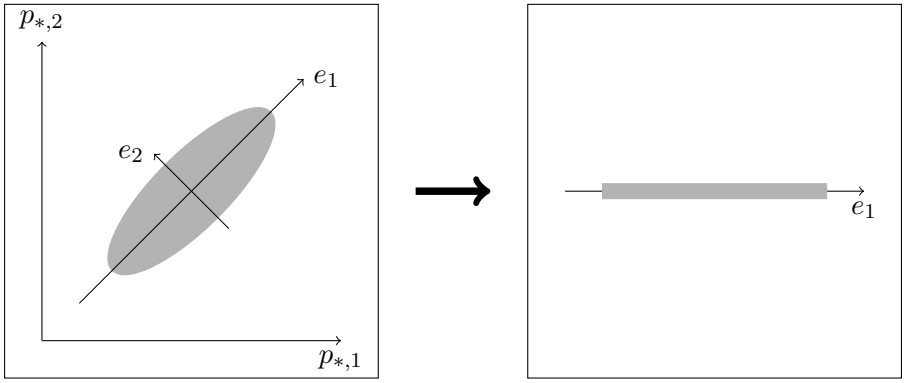


Figure 2.9: One-dimensional approximation (right) of two-dimensional data (left) by PCA. The length of the eigenvectors $\boldsymbol{e_i}$ corresponds to their eigenvalue λ_i. Adapted from [41].

2.3.1 Visual Representation of PCA

Figure 2.9 sketches the reduction of 2D data, that is indicated as a gray ellipse, into one-dimensional values. The lengths of the drawn eigenvectors $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ depict the corresponding eigenvalues. By removing the dimension $\boldsymbol{e_2}$, the data is approximated with only one dimension. The number of kept dimensions varies depending on the application.

In our work we make use of PCA without reducing dimensions. Figure 2.10 shows the same example again but without discarding the second axis. Without loss of generality we let the major axis point to the right and the minor axis point upwards. Once the data is aligned in this way, one can compare different point distributions more easily independent of their global alignment.

Higher-order features are derived by some simple calculations. When the data was aligned along a straight line, $\frac{\lambda_1}{\lambda_2}$ is very big. If we had a more circular distribution instead, both eigenvalues would have been about the same values resulting in $\frac{\lambda_1}{\lambda_2} \approx 1$.

In 3D point cloud processing this is heavily used to calculate local features. For instance one can apply PCA on a number of 100 neighboring 3D points and therefore gather information about their local structure. Despite these only-local features, other properties can also be taken into account. For instance, the previously defined eigenvalue-based features cannot distinguish horizontal and vertical point alignments. But this is needed to differentiate for example between poles and overhead power lines. One possible additional feature can therefore be the global alignment of the first principal component $\boldsymbol{e_1}$. When the data is distributed vertically, $\boldsymbol{e_1}$ will be approximately parallel to the global vertical axis. A detailed description of the features we used can be found in chapter 6.

2.3.2 Calculations in PCA

Like previously shown, PCA results in a new orthogonal coordinate frame built by the eigenvectors $\boldsymbol{e_i}$. We let $Q = \{\boldsymbol{q_1}, \boldsymbol{q_2}, \ldots, \boldsymbol{q_k}\} \subset P$ denote neighbors of a point we calculate features for. To ensure that the eigenvector with the highest eigenvalue really represents the dimension with the highest data variance, the data must be distributed around the center of the global co-

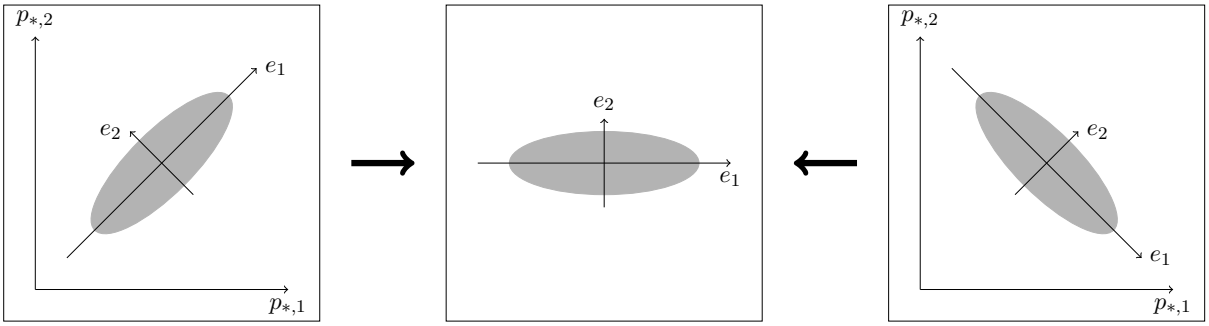


Figure 2.10: PCA without reduction of dimensions for calculating features. Both data distributions (left and right) reveal the same features with respect to their eigenvectors $\boldsymbol{e_i}$ and eigenvalues λ_i. The length of the eigenvectors corresponds to their eigenvalue.

ordinate frame. To ensure this, the mean $\bar{\boldsymbol{q}} = [\bar{q}_1 \;\; \bar{q}_2 \;\; ... \;\; \bar{q}_d]^\intercal$ is subtracted from every neighbor. This results in the $(d \times n)$ matrix $\tilde{\boldsymbol{Q}}$ according to equations 2.9 and 2.10. d denotes the points' dimensions to enable general equations.

$$\bar{\boldsymbol{q}} = \frac{\sum_{i=1}^{k} \boldsymbol{q_i}}{k} \tag{2.9}$$

$$\tilde{\boldsymbol{Q}} = \begin{bmatrix} \boldsymbol{q_1} - \bar{\boldsymbol{q}} & \boldsymbol{q_2} - \bar{\boldsymbol{q}} & \cdots & \boldsymbol{q_n} - \bar{\boldsymbol{q}} \end{bmatrix} = \begin{bmatrix} q_{1,1} - \bar{q}_1 & q_{2,1} - \bar{q}_1 & \cdots & q_{n,1} - \bar{q}_1 \\ q_{1,2} - \bar{q}_2 & q_{2,2} - \bar{q}_2 & \cdots & q_{n,2} - \bar{q}_2 \\ \vdots & \vdots & \ddots & \vdots \\ q_{1,d} - \bar{q}_d & q_{2,d} - \bar{q}_d & \cdots & q_{n,d} - \bar{q}_d \end{bmatrix} \tag{2.10}$$

Afterwards the eigenvectors of the data's covariance matrix $\boldsymbol{V}$, which is determined by the calculation shown in equation 2.11, needs to be calculated. $\boldsymbol{V}$ is a symmetrical $(d \times d)$ matrix.

$$\boldsymbol{V} = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{Q}}^\intercal \tag{2.11}$$

The relationship between the eigenvalues λ_i with their eigenvectors $\boldsymbol{e_i}$ is given by the two statements shown in equation 2.12.[57]

$$\begin{aligned} \lambda_i \boldsymbol{e_i} &= \boldsymbol{V}\boldsymbol{e_i} \\ (\boldsymbol{V} - \lambda_i \boldsymbol{I})\boldsymbol{e_i} &= 0 \end{aligned} \tag{2.12}$$

Solving the linear system of equations is done by determining the characteristic polynomial $\boldsymbol{P_V}$ (equation 2.13) and setting it to zero. The zeros λ_i correspond to the eigenvalues of $\boldsymbol{V}$ with eigenvectors $\boldsymbol{e_i}$.

$$\boldsymbol{P_V} = \det(\boldsymbol{V} - \boldsymbol{\lambda I}) = \begin{vmatrix} v_{11} - \lambda & v_{12} & \cdots & v_{1d} \\ v_{21} & v_{22} - \lambda & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ v_{d1} & \cdots & \cdots & v_{dd} - \lambda \end{vmatrix} \tag{2.13}$$

For a high number of dimension d, instead of solving the polynomial of degree d to find zeros, *Singular Value Decomposition* (SVD) [84, 113] is often used. SVD splits the covariance matrix into three matrices $\boldsymbol{E\Sigma E}^\intercal$. $\boldsymbol{E}$ here contains the eigenvectors of the covariance matrix and $\boldsymbol{\Sigma} = diag(\sqrt{\lambda_1}, \sqrt{\lambda_2}, \ldots, \sqrt{\lambda_d})$. The singular values are the square root of the searched eigenvalues λ_i of $\boldsymbol{V}$. The advantage of using SVD is that there is no need to calculate $\boldsymbol{V}$ explicitly to find eigenvalues and eigenvectors by its characteristic polynomial.[41]

2.4 Classification

2.4.1 Random Forests

Random forest (RF) classification is an ensemble method that constructs a multitude of different random trees during training. It can be seen as a special form of bagging of decisions trees.

Decision Trees

Table 2.1: Example data for building a decision tree. Possible classes are *pole/railing*, *facade*, and *ground*

Nr.	**Feature f_1** ($\frac{\lambda_1}{\lambda_2+\lambda_3}$)	**Feature f_2** ($e_1 \approx$ ***horizontal***)	**Class**
1	high	yes	pole/railing
2	high	no	pole/railing
3	low	yes	facade
4	low	no	ground

To be able to fully motivate RFs, we first have to introduce decision trees. Table 2.1 shows a small sample of training data with four instances. Each line corresponds to a 3D point which belongs to one of the classes *pole/railing*, *facade*, or *ground*. The two features shown in the table are very similar to the ones we previously described. We denote them as f_1 and f_2. f_1 takes the distribution in the three principal axes resulting from the performed PCA into account. f_2 describes the alignment of the major axis in the global coordinate frame.

For this small dataset classification rules can easily be determined: Poles and railings have a high distribution spread along only one axis. Points belonging to facades or ground are likely to have around the same distribution along their two main axes, resulting in a low value for the first feature. To be able to distinguish further between facade and ground points, one has to look at the second feature, the alignment within the global coordinate frame. The first principal

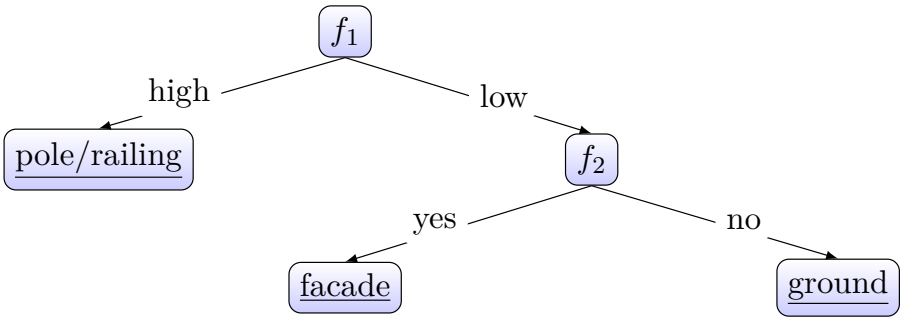


Figure 2.11: Decision tree to distinguish between the classes *pole/railing*, *facade*, and *ground*. The input contains two features (f_1 and f_2). Underlined text indicates a leaf node that returns a class prediction. Refer to table 2.1 for the underlying training data.

axis is approximately horizontal only for the class *facade*, not for *ground* points. We simplified this example by introducing only nominal/binary values. Later, numeric values are used.

A decision tree, as shown in figure 2.11, is typically automatically generated. At each level it has to be decided which feature is used for splitting. Instead of testing all possible trees, different algorithms were developed to find a good tree in less computing time. In Weka the extension *C4.5* of the standard *ID3* algorithm is implemented. The implementation is also called *J48*. It allows to also handle numeric and weighted attributes as well as missing values. For reasons of clarity we only introduce the underlying base algorithm here.

The ID3 algorithm does a greedy search through possible tree branches without backtracking [89]. At each level the locally best split is performed. In each of the resulting subtrees new split decisions are done iteratively. Only the subset of training data that fits the current position in the tree is used for further decisions.

$$H(S) = -\sum_{c \in C} p_S(c) \log_2 p_S(c) \tag{2.14}$$

Typically, *information gain* is used as a metric to determine which attribute is used for the next split. To calculate information gain, Shannon entropy first needs to be defined. This definition is shown in equation 2.14. S denotes the set of current training instances at this split node and C the set of classes in S. Furthermore $p_S(c)$ describes the proportion of elements belonging to a class $c \in C$ in the set S.

$$Gain(S, f_i) = H(S) - \sum_{v \text{ is a value of } f_i} p_S(f_i = v) H(S_{f_i=v}) \tag{2.15}$$

Now information gain can be defined as shown in equation 2.15. $f_i, i \in 1, 2, \ldots, d$ represents one possible split attribute the information gain is calculated for. $S_{f_i=v} \subset S$ denotes the subset of elements for which $f_i = v$ holds true. The according probability of an element to belong to this subset is denoted as $p_S(f_i = v) = \frac{|S_{f_i=v}|}{|S|}$.

At every split, the information gain is maximized by first calculating $Gain(S, f_i)$ for all possible features f_i. Then the attribute with the highest information gain is picked. Respectively, one can say (see equation 2.15) that the entropy within the resulting subtrees is minimized. Each subtree's entropy is weighted by it's size defined by the number of supporting training samples. This metric homogenizes the subsets of data with each split.[116]

Algorithm 2 shows all steps of the ID3 algorithm to gain a decision tree from a given dataset S.[50] In our example, presented in table 2.1, we start with all training data S and $R = \{f_1, f_2\}$. As we have three different classes present in S and there are still remaining split attributes ($|R| > 0$), we can search for the best split attribute. For this we have to calculate $Gain(S, f_1)$ and $Gain(S, f_2)$ as shown in equation 2.16 et seqq. $H([h_1, h_2, ...])$ denotes the entropy of a distribution where the first class appears h_1 times, the second one h_2 times, et cetera. Therefore it is $p_S(c_i) = \frac{h_i}{\sum_j h_j}$.

Algorithm 2 ID3 algorithm to create decision trees.

Input: S: Set of training data, R: Set of possible split-attributes $f_1, f_2, \ldots, f_d$ (class excluded)
Output: Decision tree

1: **if** $|S| = 0$ **then** *// Training set empty.*
2: **return** Error node
3: **end if**
4: **if** $\forall s \in S$ belong to same class c **then** *// We do not need further splits.*
5: **return** Leaf node with result c
6: **end if**
7: **if** $|R| = 0$ **then** *// No attributes for splitting left. We have to predict a class now.*
8: **return** Leaf node with most common class c in S
9: **end if**
10: $f_{best} \leftarrow$ feature $f_i \in R$ with maximum $Gain(S, f_i)$
11: $R \leftarrow R \backslash \{f_{best}\}$
12: $\{S_{f_{best}=v_i}, i \in \{1, 2, ..., m\}\} \leftarrow m$ partitions of S with $S_{f_{best}=v_i} = \{s \in S | s$ has attribute value $f_{best} = v_i\}$
13: Iteratively run algorithm with training data $S_{f_{best}=v_i}$ and new, smaller set of possible split-attributes R.
14: **return** Decision node with connection to the m subtrees that we created iteratively

$$
\begin{aligned}
H(S) &= -\sum_{c \in C} p_S(c) \log_2 p_S(c) \\
&= -[p_S(pole/railing) \log_2 p_S(pole/railing) + p_S(facade) \log_2 p_S(facade) \\
&\quad + p_S(ground) \log_2 p_S(ground)] \\
&= -[2/4 \log_2 2/4 + 1/4 \log_2 1/4 + 1/4 \log_2 1/4] \\
&= -[0.5 \cdot (-1) + 0.25 \cdot (-2) + 0.25 \cdot (-2)] = 1.5
\end{aligned} \tag{2.16}
$$

$$
\begin{aligned}
Gain(S, f_1) &= H(S) - \sum_{v \in \{high, low\}} p_S(f_1 = v) H(S_{f_1=v}) \\
&= H(S) - [(p_S(f_1 = high) H(S_{f_1=high})) \\
&\quad + (p_S(f_1 = low) H(S_{f_1=low}))] \\
&= 1.5 - [(1/2 H([2])) + (1/2 H([1, 1]))] \\
&= 1.5 - [(1/2 \cdot 0) + (1/2 \cdot 1)] = 1.5 - 0.5 = 1
\end{aligned} \tag{2.17}
$$

$$
\begin{aligned}
Gain(S, f_2) &= H(S) - \sum_{v \in \{yes, no\}} p_S(f_2 = v) H(S_{f_2=v}) \\
&= H(S) - [(p_S(f_2 = yes) H(S_{f_2=yes})) \\
&\quad + (p_S(f_2 = no) H(S_{f_2=no}))] \\
&= 1.5 - [(1/2 H([1, 1])) + (1/2 H([1, 1]))] \\
&= 1.5 - [(1/2 \cdot 1) + (1/2 \cdot 1)] = 1.5 - 1 = 0.5
\end{aligned} \tag{2.18}
$$

As it holds $Gain(S, f_1) > Gain(S, f_2)$, the first split attribute is chosen to be f_1.

On the second level, two subtrees have to be created. In the left subtree (see figure 2.11) that handles all instances where $f_1 = high$, all points belong to the class *pole/railing*. Therefore no further splitting is needed. A leaf node predicting this class is created.

On the other side, where $f_1 = low$ more than one class is present. Normally the best split attribute is chosen now by calculating the information gain for all possible splits. In this example f_2 with an information gain as shown in equation 2.19 is the only choice for further splitting. A new split node is created.

On the third level no remaining split attributes are available, so leaf nodes are created. The tree fits the given training data perfectly.

$$\begin{aligned} Gain(S_{f1=low}, f_2) &= H(S_{f1=low}) - \sum_{v \in \{yes,no\}} p_{S_{f1=low}}(f_2 = v) H(S_{f1=low, f2=v}) \\ &= -(1/2 \log_2 1/2 + 1/2 \log_2 1/2) - (1/2 H([1]) + 1/2 H([1])) \\ &= 1 - [(1/2 \cdot 0) + (1/2 \cdot 0)] \\ &= 1 - 0 = 1 \end{aligned} \tag{2.19}$$

RFs in Contrast to Decision Trees

As mentioned before, random forests are a special form of bagging of decisions trees. During training, a single decision tree is created in each iteration. We will use the definition of [18] for our work with RFs as it is very common and implemented in Weka.

Bagging, which means *bootstrap aggregating*, is used to improve accuracy and stability of classification and regression algorithms. Multiple training sets are sampled from the whole set of training data. They all have the size of the original training set. By sampling uniformly with replacement, $1 - \frac{1}{e} \approx 63.2\%$ of each subset are unique examples while the rest are duplicates. Each set is then used to train one model. At the end, the output of all models is averaged (regression) or used for voting (classification).[17]

When thinking about the previously described ID3 algorithm one might wonder how multiple decision trees, trained on different training subsets, would look like. The probability of building the same tree or very similar ones again and again is very high as all the uniformly sampled sets present the original data distribution. The specialty of random forests compared to just bagging of decision trees is that at every level of the tree, only a subset of all attributes is taken into account for splitting. To allow the algorithm to create more varying trees, typically only $\lceil \sqrt{d} \rceil$ or $\lfloor \log_2(d) + 1 \rfloor$ (Weka) of all d features are taken into account when performing split decisions.

Parameters in our RF Implementation

In this section we describe the two parameters that were most important to us when using random forests. Interested readers may also consult [97] which describes an early study on the effect of parameter selection.

The number of trees in a forest, is a main parameter when constructing a RF. It has to be carefully chosen by the user as it strongly influences the performance. Speaking about

generalization error, [18] provides a proof for its convergence and the existence of an upper bound for a growing number of trees. However, the question if it is feasible to train as many trees as computationally possible is discussed for a long time. Other work showed, that increasing the number of learners not necessarily increases the overall performance but may impact it negatively. [40] illustrates this in a simple example of bagging. [86] has shown in a benchmark study on over 300 datasets that the first 100 trees in a forest achieve the highest performance improvement. In [1] different random forest implementations were tested on five datasets. The average classification accuracy in all their tests was within a one percent range when changing the parameter for the number of trees from 100 to 2,000. Meanwhile, using 100 trees has become a default setting for many random forest implementations. To be comparable to other work, we stick to this specification in most of our experiments.

Another important parameter is the tree depth. It limits the length of a path from the root to all leaf nodes. A high depth results in a big number of splits in each tree. Each tree will start to overfit on the training data. Small depths in contrast will lead to underfitting. Generally each tree should be able to predict all classes of a data set which gives a constraint of $d_t \geq \lceil \log_2 |C| \rceil - 1$ for a number of $|C|$ classes. Often this parameter is optimized together with the number of trees in a grid search.

Performance Evaluation Strategies

In [85] two typical evaluation strategies for random forests are summarized.

The first, k-fold cross validation, is very typical for many learning algorithms in machine learning. k, which is often chosen between 2 and 10, is the number of folds the training data is randomly split into. Validation is done by learning k classifiers, here random forests, on $k-1$ folds each. Each learner ignores an other fraction of the training data that is then used to evaluate this classifier. Averaging the results provides a good estimation of the later performance on other unseen data.[98]

When using random forests or other bagging techniques, out-of-bag observations are another strategy to evaluate the training. For this, only a randomly chosen part of the training data is used for learning each single model, here random tree, in the ensemble. Afterwards all the training examples are classified by only the classifiers that did not use this example for training.

However, these strategies did not seem very suited for our application. In our approach we calculate features for optimal local neighborhoods. In the preferred case this neighborhood will contain roughly the same points for many neighboring 3D points representing the same object. This results in nearly the same feature values for all of these points whereas other instances of similar objects will result in diverse values. For instance, the features of two different cars, both in one scan of a parking lot, might be very different due to various factors. They are influenced by the distance and angle between the laser scanner and the car. Also other objects might occlude parts of one car as discussed before.

For these reasons and because these methods need a higher computation time, we decided to use the classical *hold-out* strategy where the dataset is split into training, validation, and test data. Between these sets all scans should be taken at different scenes. Where this was not feasible, at least very differing scanner positions were preferred.

Handling of Highly Imbalanced Data

The data that is used to train the random forests later is highly imbalanced. Tables 4.2 and 4.4 show factors of 100 and 1,000 respectively between different classes – even after preprocessing and deletion of very small classes. This is caused by the natural structure of the scenes. While ground and facades are very dominant in urban scenes, motorcycles, bushes, or trash cans are seen less often. Even if many instances of these infrequent objects are scanned, their small footprint results in a discrimination of their share in the 3D point cloud.

Learning a RF directly on this imbalanced data results in a classifier that has a bias towards these predominant classes.[22] For instance, a classifier only correctly predicting *facade* and *ground* on the Paris-rue-Madame dataset (see table 4.4) is able to achieve approximately 90% accuracy without ever predicting another class. Of course this behavior is undesirable even if the overall accuracy leads to believe the learner successfully trained.

A general approach to solve this problem is sub-sampling. Two variants are very common. *Up-sampling* means that the training data for minority classes is extrapolated. Usually this is done by sampling each class equally often, n_{max} times, where n_{max} is the size of the dominating class. On the other hand, there is *down-sampling*. Each class is only sampled n_{min} times where n_{min} is the number of training examples belonging to the smallest class. A big number of instances of the dominant classes are skipped during training. Further research with imbalanced data and about the fact that down-sampling is often preferred over up-sampling are found in [33, 59, 68].

In this thesis we stick to that recommendation for two more application dependent reasons. First, many data points of the big classes are redundant. Neighboring points of a class, for instance man-made ground, will result in nearly the same features. Throwing away some of this data is expected not to hurt the classification performance too much. Second, up-sampling small classes results in over-fitting on specific instances of objects. While traffic signs can look very different, the classifier might only see a small subset of possible signs. Showing this training examples again and again will lead to over-fitting on these instances. We favor to learn a classifier that will also notice other unseen traffic signs.

Influence of Class Imbalances During Evaluation To be comparable, we decided to mainly use the four common measures *accuracy*, *precision*, *recall*, and F_1 *score* as shown in equation 2.20 et seqq. However, since we publish high-detailed results, many other measures can be derived by the reader.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.20}$$

$$Precision = \frac{TP}{TP + FP} \tag{2.21}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.22}$$

$$F_1 = 2\frac{Precision \cdot Recall}{Precision + Recall} \tag{2.23}$$

Figure 2.12 illustrates the division of data in *false negatives* (FN), *true negatives* (TN), *true positives* (TP), and *false positives* (FP).

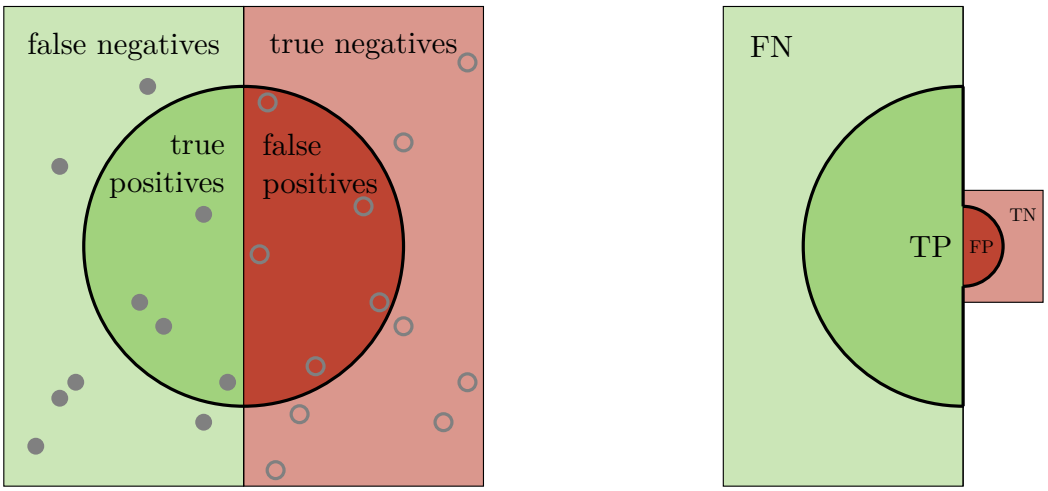


Figure 2.12: Visualization of *TP*, *FP*, *TN*, and *FN* for balanced (left) and imbalanced (right) data. Left: All gray, filled circles belong to one specific class to predict. The classifier's positive class predictions are indicated by the big circle in the middle. Right: Impact of class imbalances. While the relative classification performance within the class inliers (TP and FN) and outliers (TN and FP) did not change, a comparison between them, for instance TP and FP, is biased. Adapted from [112].

When comparing our results on highly imbalanced datasets the following has to be kept in mind.

Precision tends be low for small classes As the number of class outliers is high, the probabilty of classifying an outlier wrongly (FP) is also high. Since additionally the number of true positives (TP) is low, this leads to a low precision for small classes.

Precision tends to be high for dominant classes Due to the fact that the number of class inliers is higher than the number of outliers (compare to figure 2.12), true positives might also dominate false positives. This results in a high precision value.

Recall gives a better understanding As recall only focuses on true positives and false negatives, class imbalances have no effect in the evaluation. Therefore this metric is more meaningful in our context.

The F_1 measure is a specialization ($\beta = 1$) of the more general F_β measure. It is characterized by the harmonic mean of precision and recall. Equations 2.23 and 2.24 depict these measures.[23]

$$F_\beta = \frac{(\beta^2 + 1.0) \cdot \mathit{Precision} \cdot \mathit{Recall}}{\beta^2 \cdot \mathit{Precision} + \mathit{Recall}} \quad (0 \leq \beta \leq +\infty) \tag{2.24}$$

Of course many other evaluation metrics are imaginable. There are cases of use for all of them and it depends on the exact application and importance of wrong classifications per class which one to pick. We stick to the recommendation of [60] using precision and recall and also print confusion matrices for some of our experiments in the appendix to allow readers to calculate other metrics than the ones used in our comparison.

2.4.2 Support Vector Machines

Support vector machines (SVMs) [110] are another popular classification technique. Their concept was introduced by Vladimir Vapnik in 1979 and later heavily improved by Bernhard Schölkopf [96]. In the simplest form, a SVM perfectly separates positive and negative examples by a hyperplane. For d-dimensional data this separating hyperplane is given by $(d-1)$-dimensions. This plane is chosen in a way that the margin between both classes is maximized. A small number of examples closest to the plane defines it. They are called *support vectors*. Figure 2.13 shows this on two-dimensional example data.

The output o_i of such a linear SVM can be computed as shown in equation 2.25, were $\boldsymbol{\omega}$ is the normal vector to the splitting hyperplane and $\boldsymbol{x_i} = [x_{i,1}\ x_{i,2}\ \dots\ x_{i,d}]^\intercal, i \in \{1, \dots, n\}$ denotes the input vectors. A displacement of the hyperplane is done by using the bias $b \in \mathbb{R}$.

$$o_i = \boldsymbol{\omega} \cdot \boldsymbol{x_i} + b \tag{2.25}$$

The hyperplane itself is therefore given by $o = 0$ while the closest points lie on the planes with $o = \pm 1$. The resulting margin m is given by equation 2.26.

$$m = \frac{1}{||\boldsymbol{\omega}||} \tag{2.26}$$

A SVM, however, finds the optimal plane with the maximum margin. The corresponding optimization problem is shown in equation 2.27. $t_i \in \{1, -1\}, i \in \{1, ..., n\}$ denotes the corresponding target output. Multi-class classification can be done with multiple hyperplanes by following a *one-vs-one* or *one-vs-rest* approach. For this thesis we will not dive deeper into this.

$$\begin{aligned} \min_{\boldsymbol{\omega} \in \mathbb{R}^n, b \in \mathbb{R}} \quad & \frac{1}{2} ||\boldsymbol{\omega}||^2 \\ \text{subject to} \quad & t_i(\boldsymbol{\omega} \cdot \boldsymbol{x_i} + b) \geq 1, i \in \{1, ..., n\} \end{aligned} \tag{2.27}$$

In 1995, a modification was introduced that allowed SVMs to also work with not linearly separable classes.[26] Wrong predictions are allowed but penalized. This led to equation 2.28 with slack variables ζ_i that permit margin failure and C as a penalty parameter for the error term to regulate the trade-off between a wide margin and a small number of margin failures.

$$\begin{aligned} \min_{\boldsymbol{\omega} \in \mathbb{R}^n, b \in \mathbb{R}, \zeta_i \in \mathbb{R}^+} \quad & \frac{1}{2} ||\boldsymbol{\omega}||^2 + C \sum_{i=1}^{n} \zeta_i \\ \text{subject to} \quad & t_i(\boldsymbol{\omega}^\intercal \boldsymbol{x_i} + b) \geq 1 - \zeta_i, i \in \{1, ..., n\} \\ & \zeta_i \geq 0, i \in 1, ..., n \end{aligned} \tag{2.28}$$

Typically, this optimization is solved using the dual, Lagrangian, form. Additionally, a kernel function K is often used. This function maps non-linear separable input data into a higher dimensional space where a hyperplane can be found to separate the samples. This allows the SVM to not only split linearly.

Altogether this leads to an output of the SVM as shown in equation 2.29, and the optimization problem shown in equation 2.30 with Lagrange multipliers α.[83]

$$o = \sum_{j=1}^{n} t_j a_j K(\boldsymbol{x_j}, \boldsymbol{x}) + b \tag{2.29}$$

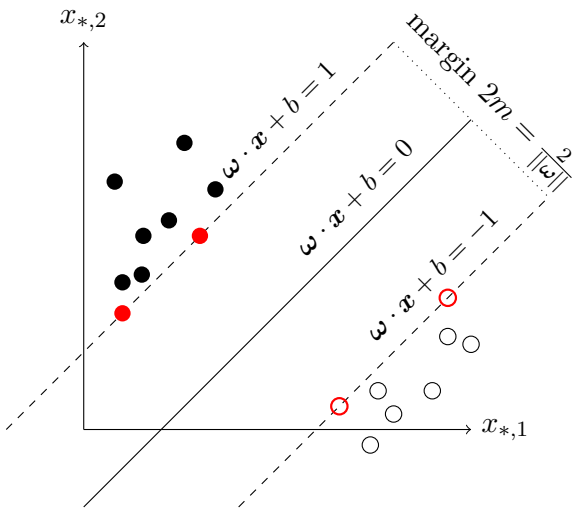


Figure 2.13: Hyperplane with maximized margin found by a SVM. Suppport vectors are marked in red.

$$\begin{aligned} \min_{\alpha} \Psi(\alpha) = \min_{\alpha} \quad & \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} t_i t_j K(\boldsymbol{x_i}, \boldsymbol{x_j}) \alpha_i \alpha_j - \sum_{i=1}^{n} \alpha_i \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \forall i \\ & \sum_{i=1}^{n} t_i \alpha_i = 0 \end{aligned} \tag{2.30}$$

Typical kernel functions, with γ, r, and d as kernel parameters, are:

linear: $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \boldsymbol{x_i}^\intercal \boldsymbol{x_j}$

polynomial: $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = (\gamma \boldsymbol{x_i}^\intercal \boldsymbol{x_j} + r)^d$

radial basis function (RBF): $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \exp(-\gamma \left|\left|\boldsymbol{x_i} - \boldsymbol{x_j}\right|\right|^2)$

sigmoid: $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \tanh(\gamma \boldsymbol{x_i}^\intercal \boldsymbol{x_j} + r)$

Learning of a SVM is nowadays often done by the *Sequential Minimal Optimization* (SMO) algorithm. It splits the large quadratic programming optimization problem into small subproblems. The memory amount required for SMO is linear in the size of the training set, while its runtime is between linear and quadratic.[83] We will skip more details here. Readers may consult [83] for more information about SVMs and the SMO algorithm. [14, 26] also provide more information about SVMs and especially the here described C-SVCs. An adaption, called SVR, can also be used for regression tasks.

2.4.3 Artificial Neural Networks

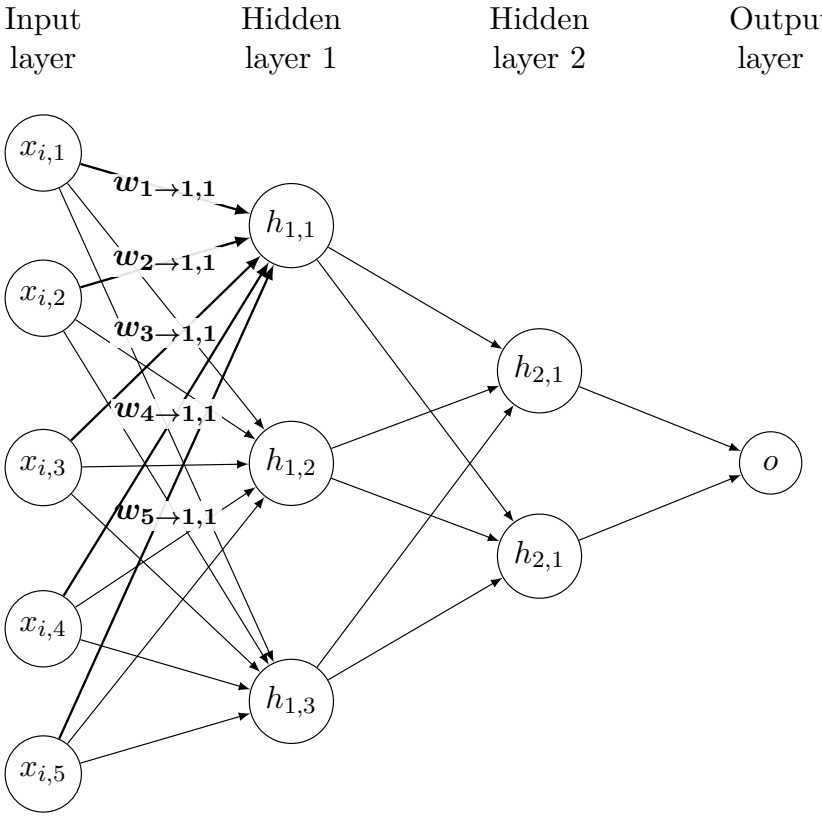


Figure 2.14: Example of a feed-forward multilayer perceptron containing two hidden layers. The weights that are used to calculate the value of $h_{1,1}$ given the input x_i are highlighted.

Artifical neural networks (ANNs) are a method in machine learning that is inspired by biological neural networks. They are successfully applied in many large real world problem areas like natural language processing [25], image classification [24, 58], human action recognition [53], and physics [74]. A general network of connected nodes, also called *artificial neurons*, is trained. In a classification task, the network learns complex relations by just analyzing labeled training examples – specific handcrafted features are not required as an intermediate step.

The most-known and still easy to understand neural network architecture is the *feed-forward multilayer perceptron*. It consists of one input layer, a number of hidden layers, and one output layer. Each vertical layer of neurons is fully connected with the adjacent layer. The vertices in the graph form an acyclic graph. An example with two hidden layers is depicted in figure 2.14.

The computation of a neuron's output is done by summing up the weighted values of the preceding neurons. The weight of a connection between node i and node j is given by $w_{i\to j}$. Also, a bias term b_j is added to the sum. Nonlinearity of the system is achieved by an additional

activation function σ. For instance, the output of the neuron $h_{1,1}$ in the figure is calculated as shown in equation 2.31 with $x_i = [x_{i,1} \;\; x_{i,2} \;\; \dots \;\; x_{i,5}]^\intercal$ denoting a five-dimensional input.

$$h_{1,1} = \sigma(\sum_{i=1}^{5}(w_{i\to 1,1} \cdot x_i) + b_{1,1}) \tag{2.31}$$

The choice of an activation function σ depends on the application. Very popular ones are, for instance, the *ReLU* (equation 2.32) and the *sigmoid* function (equation 2.33).

$$ReLU(x) = \begin{cases} 0 & x \leq 0 \\ x & \text{else} \end{cases} \tag{2.32}$$

$$sig(x) = \frac{1}{1 + e^{-x}} \tag{2.33}$$

Given enough artificial neurons, a feed-forward multilayer neural network can represent any function. When used in a classification task, the output layer usually consists of multiple neurons $o_1, o_2, \dots, o_{|C|}$ for a number of $|C|$ classes. By applying the *softmax* function shown in equation 2.34 on all of these neurons, the probabilities of the input belonging to each of the classes is calculated. Usually, the one with the highest probability is picked as the predicted class.

$$softmax(o)_i = \frac{e^{o_i}}{\sum_{j=1}^{|C|} e^{o_j}} \tag{2.34}$$

Training a Neural Network At the beginning, all weights and biases are randomly initialized. The network will therefore often output the wrong value/class for an input. To correct this, a high number of training examples is needed and the so-called *backpropagation algorithm* [47] is applied. The training process iterates two steps.

First, a forward pass is performed. The network calculates an output with its current weights for given input data. The error is defined by a loss function describing the difference between the target output and the actual output.

Second, the error is backpropagated to all the previous nodes in the network. Calculating the derivatives with respect to all the weight and bias parameters allows to reduce the error by following its negative gradient. By repeating both steps, the error is minimized step-by-step. A user-defined *learning rate* determines the magnitude of changes in the network and therefore the time needed until its convergence to a state with local minimal error.

In reality, ANNs are often huge. For instance, networks used for the classification of 2D images often contain over a dozen layers with multiple thousand neurons each which all need to be trained. While in the recent years much work was done to be able to describe the internally learned structure of neural networks [75], they are still considered as *black boxes*.[81]

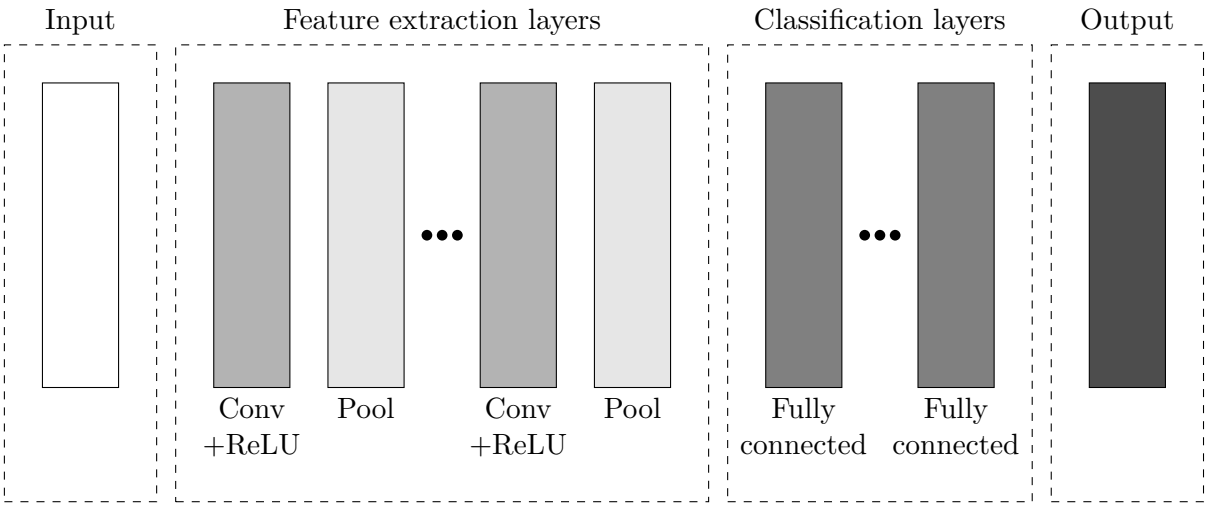


Figure 2.15: Structure of a convolutional neural network (CNN). Adapted from [81].

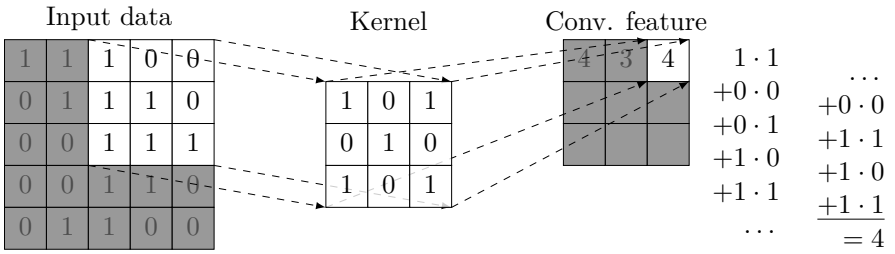


Figure 2.16: Illustration of a convolution layer. Adapted from [81].

Convolutional Neural Networks Convolutional neural networks (CNNs) are one major architecture of deep networks. As KPConv, the later applied deep learning approach, heavily depends on them, they are introduced here.

The goal of CNNs is to learn higher-order features of the data via convolutions. Often, they are applied for object recognition in images as they achieve very high accuracies in this task. They are capable of identifying faces, individuals, street signs, and many other aspects of visual data. Inspired by the visual cortex, CNNs exploit the spatial arrangement of input data. The perception in small subregions is tiled together to cover the entire visual field.

A CNN typically consists of an input layer – taking a two-dimensional input image – repeatedly followed by convolution and pooling layers. These layers are used to extract higher order features that are then fed into multiple fully connected layers for classification. This common structure is depicted in figure 2.15.

Every pooling layer heavily reduces the data's complexity. Most of the time *max-pooling* with a kernel size of 2×2 is done. This means that a sliding window with a height and a width of two pixels is moved over the data with a stride length of two and only the maximum of the four seen values is kept.

The convolution layers, however, detect local patterns. This mathematical operation describes a rule for how to merge two sets of information. A kernel, for instance with a size of 3×3 pixels, is slid over the data. Usually a stride of one pixel is chosen, but other values are possible. Figure 2.16 depicts an example for a monochrome image with binary pixel values. The kernel was already slid over the first row of data. At its current position the calculated convolution feature is 4. It is the sum of the input data weighted by the corresponding kernel values. The exact calculation is also shown in the figure. A more mathematical definition with respect to 3D points is shown in section 6.4.2. In the training process, the kernel values take over the function of the weights of normal layers. They are adjusted during backpropagation.[81]

Part I

Tools and Datasets

Chapter 3

Used tools

3.1 3DTK - The 3D Toolkit

3DTK - The 3D Toolkit [80], formerly known as *Slam6D*, is a powerful toolkit for 3D point cloud processing mainly written in C++. It is available online at `http://three-dtk.de`. The project started at the *GMD* (*"Gesellschaft für Mathematik und Datenverarbeitung"*) in 2000 and its featureset was expanded since then by the *Fraunhofer Institute*, the *University of Osnabrück*, and the *Jacobs University Bremen*. Since 2013, it is mainly maintained by the *Institute of Computer Science* at the *University of Würzburg*.

The main parts of 3DTK are open source (GNU GPLv3). These are[79]:

show A fast 3D viewer. It allows smooth movement in large 3D point clouds via mouse and/or keyboard by frustum culling and a dynamic point reduction. *Show* by default only renders as many points as possible to keep 20 frames per second. It is also capable of taking colored screenshots or videoclips while flying through the scene. Some additional features were developed during the work on this thesis. They allow *show* to handle and display 3D point clouds with class labels. A screenshot is shown in figure 3.1.

slam6d A high-accurate algorithm for 6D simultaneous localization and mapping (SLAM).

scan_red Reduction of 3D point clouds. It can also be used to generate panorama images.

scanner Used to build 3D models out of videos with moving laser lines.

Figure 3.1 shows the graphical user interface (GUI) of the 3D viewer *show*. The GUI is split into three parts. On the right, the selection pane allows the user to change general settings and perform common tasks. One can for instance adjust the point size or the fog density there. The choice of a color map allows an improved representation of the point cloud. Actions available in this pane are for example the recovery of previously saved poses and the creation of screenshots and video animations.

The changes affect the point cloud view in the main window where a user can navigate through the cloud by mouse and keyboard commands. Additional controls are available in the lower pane. Also the view mode can be toggled and additional cameras for 3D animations can

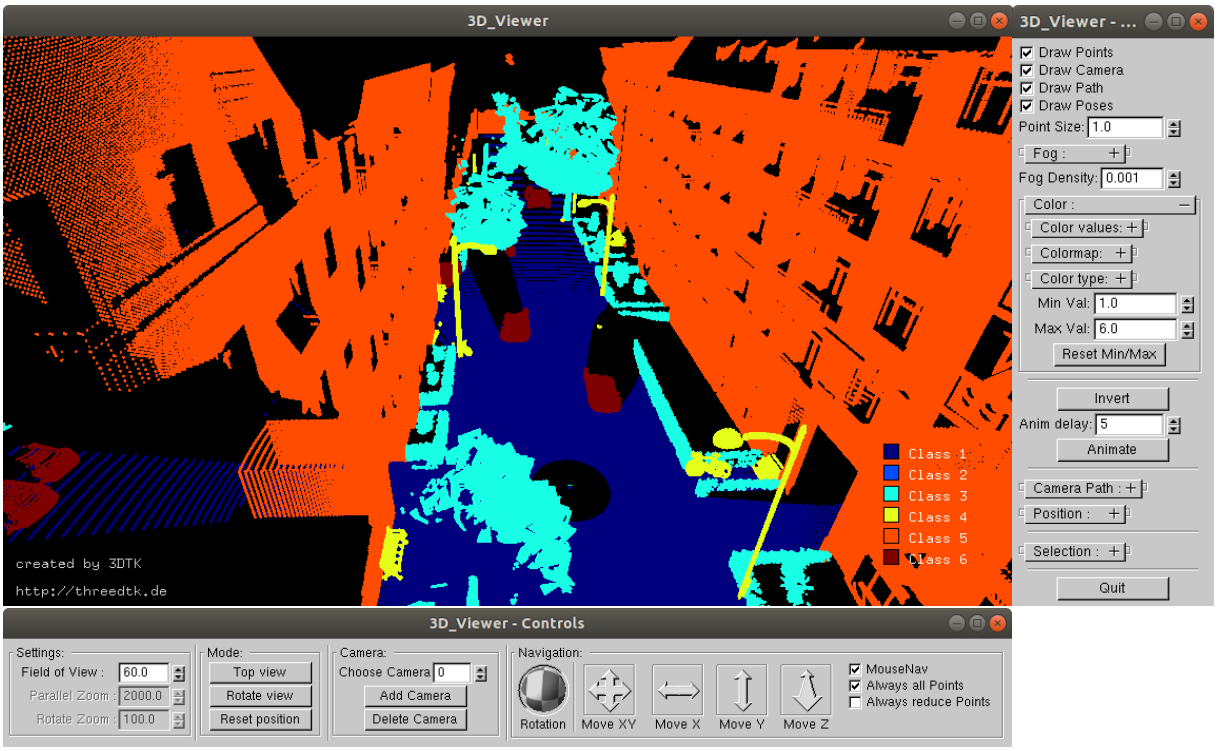


Figure 3.1: User interface of 3DTK's 3D viewer *show*. The imported point cloud was created with Blender2Helios (chapter 5) in an artificial scene of Hamburg. Thanks to *Prof. Dr. Bernhard Höfle* for providing the underlying Unreal Engine scene [56].

be created there. The most important switches are, however, *MouseNav*, *Always all Points*, and *Always reduce Points*. In *MouseNav* mode, the user navigates through the point cloud via mouse movements. If this mode is deactivated, the mouse can be used to select single points of the cloud. These are then printed on the command line or can later be exported as a partial cloud. The other two switches are used for the dynamic reduction of the cloud. To take high quality screenshots and videos, one enables *Always all Points* to always render the full point cloud. This allows 3DTK's GUI to fall under the frame refresh limit during calculations. For hardware with low performance the *Always reduce Points* flag can improve the user experience. In normal mode, where both options are disabled, 3DTK shows as much points as possible while refreshing with at least 20 fps.

One highlight of 3DTK is its efficient octree implementation already described in section 2.2.1. Each node can be represented by only 8 bytes or even 7 bytes for octrees with constant depth. Therefore 3DTK supports very large 3D point clouds.

The toolkit also uses libraries under BSD or similar licenses like *Boost*, *ROS*, and *OpenCV*. So the toolkit may also be used in commercial software.[80] Due to the simplicity of building software on top of 3DTK, we implemented the algorithms to calculate features of 3D points based on this toolkit. Details follow in part II. More work in which this powerful toolkit was used can be found in [12, 13, 34].

File formats and coordinate systems 3DTK supports many different file formats. The most basic ones are *xyz* and *uos*. For both xyz and uos there are enhanced formats that also include color information, reflectance values, normals, or temperature data. While all xyz-based formats expect the data to be in a right-handed frame of reference, uos-formats expect data points in a left-handed frame as depicted in figure 3.2. Although 3DTK works mainly without units, distances in xyz-files are expected to be measured in meters, whereas distances in uos-files are conventionally given in centimeters. This yields the conversation functions shown in equation 3.1 for the point $\boldsymbol{p} = [xyz_1, xyz_2, xyz_3]^\intercal$ in xyz-format and the point $\boldsymbol{q} = [uos_1, uos_2, uos_3]^\intercal$ in uos-format.

$$
\begin{aligned}
convert_{xyz\to uos}(\boldsymbol{p}) &= convert_{xyz\to uos}(\begin{bmatrix} xyz_1 \\ xyz_2 \\ xzy_3 \end{bmatrix}) = 100 \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} xyz_1 \\ xyz_2 \\ xzy_3 \end{bmatrix} = 100 \begin{bmatrix} -xyz_2 \\ xyz_3 \\ xyz_1 \end{bmatrix} \\
convert_{uos\to xyz}(\boldsymbol{q}) &= convert_{uos\to xyz}(\begin{bmatrix} uos_1 \\ uos_2 \\ uos_3 \end{bmatrix}) = \frac{1}{100} \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} uos_1 \\ uos_2 \\ uos_3 \end{bmatrix} = \frac{1}{100} \begin{bmatrix} uos_3 \\ -uos_1 \\ uos_2 \end{bmatrix}
\end{aligned}
\tag{3.1}
$$

3DTK works internally completely with points converted to uos. This needs to be taken into account later when calculating features. The second axis is pointing upwards in this coordinate system.

During the work for this thesis two additional file formats were implemented. These are *xyzc* and *uosc*. These are also ASCII files with coordinate conventions as previously presented. Each point, however, also contains an additional integer value corresponding to the points' class. For the best user experience in the *show* viewer, the class labels for $|C|$ classes should be chosen in steps of one, e.g. $C = \{1, 2, 3, ..., |C|\}$. This allows 3DTK to create a reasonable color legend. Instead, for scans with over 20 classes a color bar is created automatically.

(a) Axes in right-handed xyz format. Values are measured in meters.

(b) Axes in left-handed uos format. Values are measured in centimeters.

Figure 3.2: Alignment of coordinate frames in 3DTK.

3.2 Weka 3

Weka (*Waikato Environment for Knowledge Analysis*) is an open source (`https://www.cs.waikato.ac.nz/ml/weka`) machine learning software written in Java that was released under GNU GPLv3. Staff of the *University of Waikato* is maintaining the project and currently working on version 3.9. Throughout this thesis, the most current stable version, 3.8.3, was used. Figure 3.3 shows its main GUI window.[32, 117]

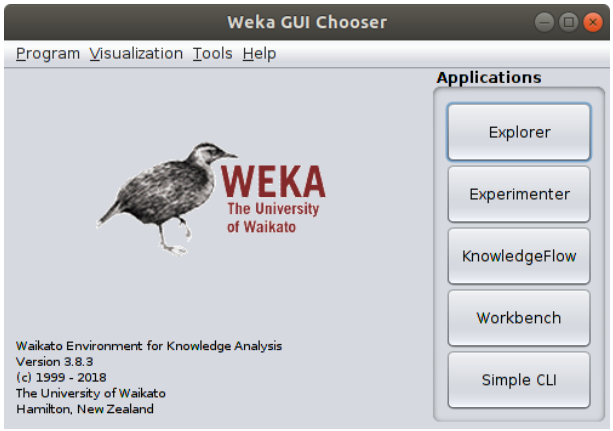


Figure 3.3: Weka's main window: The GUI Chooser.

In figure 3.4 the *Preprocessing* tab of the *Weka Explorer* is shown. It is the main tool we used for the classification process in this research. Loading, displaying, and altering training data is possible there. Visualizations are rendered automatically to provide a quick summary of the data. In the shown example the distribution of the calculated feature *planarity* between all classes (indicated by different colors) is shown. Weka can be used for

- Data Preparation,
- Classification,
- Regression,
- Clustering,
- Association Rules, and
- Visualization.

With [116] there is also a book available that explains many aspects of machine learning, always with examples and implementations in Weka.

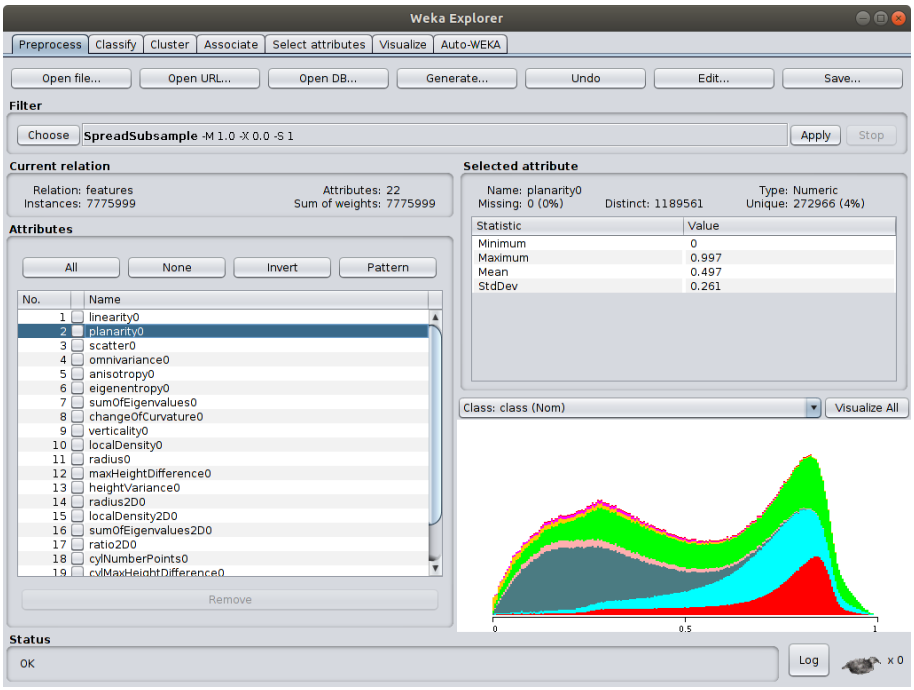


Figure 3.4: The *Weka Explorer* for easy data preprocessing.

3.2.1 The ARFF File Format

The Attribute-Relation File Format, short ARFF, standardizes ASCII text files containing a list of instances sharing a set of attributes. It is the main data format that is used by Weka. An ARFF file consists of two distinct sections, a header and a data section. ARFF is completely defined as ANTLR (*ANother Tool for Language Recognition*) v3 grammar in the document `https://waikato.github.io/weka-wiki/files/arff.g`.

The name of the relation and the list of attributes including their types is listed in the header. Percent signs indicate comments. The following lines are an example describing a labeled 3D point cloud where each data point consists of three real numbers (x, y, z) and a nominal class label.

Header section of an ARFF file:

```
% Example 3D point cloud
@RELATION pointcloud

@ATTRIBUTE x NUMERIC
@ATTRIBUTE y NUMERIC
@ATTRIBUTE z NUMERIC
@ATTRIBUTE class {Ground, Building, Car}
```

The actual data is then joined as data section. The following example indicates five labeled 3D points while the last point misses one attribute value.

Data section of an ARFF file:

```
@DATA
8.9,3.4,5.4,Car
7.2,7.8,0.3,Ground
2.4,7.8,9.0,Building
2.5,7.7,9.0,Building
% Also missing values can be represended. See '?' in the next line
2.6,?,9.1,Building
```

More details, including the representation of sparse ARFF files and how to introduce instance weights in ARFF files, can be found in [3].

3.2.2 Support Vector Machines in Weka

The prefered way to use support vector machines in Weka is to additionally install the *LibSVM* package [21]. Installation can be easily done in Weka's build-in package manager. LibSVM, which is also available at `https://www.csie.ntu.edu.tw/~cjlin/libsvm`, is a wrapper class for the more powerful *LibSVM tools* library. Nevertheless, most of the features are present and the easy integration in a graphical tool like Weka allows for a broad usage of its main features.[21] Besides LibSVM's Java and Weka support, there are also interfaces for Matlab/Octave, R, Python, Node.js, JavaScript, .NET, C#, PHP, GO, and many more.

We decided on using version 1.0.10 of LibSVM with our Weka installation for our SVM experiments. This was the latest version available in 2020 when this thesis was written. It has to be noted that there is also a *LibLinear* package which allows better runtimes while only supporting SVMs with a linear kernel function. However, as linear SVMs were quickly outperformed in terms of classification accuracy by the other methods, we did not put any effort into speed improvements.

3.3 Helios

Helios, the *Heidelberg LiDAR Operations Simulator*, is a software package for interactive real-time simulation and visualization of laser scanning surveys. Terrestrial, mobile and airborne laser scans are supported. The framework is split up into a core component and extension modules, all written in Java and available at `https://github.com/GIScience/helios`.[5]

Figure 3.5 summarizes the different XML files needed to configure and run Helios. For our work we wanted to create 3D scans of very realistic scenes. While it is was easy to create the more general and short *Survey.xml*, *Scanner.xml*, and *Platform.xml*, writing compatible *Scene.xml* files turned out to be a very time consuming task in the preparation for a laser survey. A XML node for each object within the scene had to be created with a reference to its object file. These had to be exported as a supported file type for every single item beforehand. Naturally, translation and rotation needed to be defined in the XML file as well. No simple way, like a level editor, was available for this task. We therefore decided to create a new tool called Blender2Helios. This software provides an interface between the well known 3D software Blender and the LiDAR

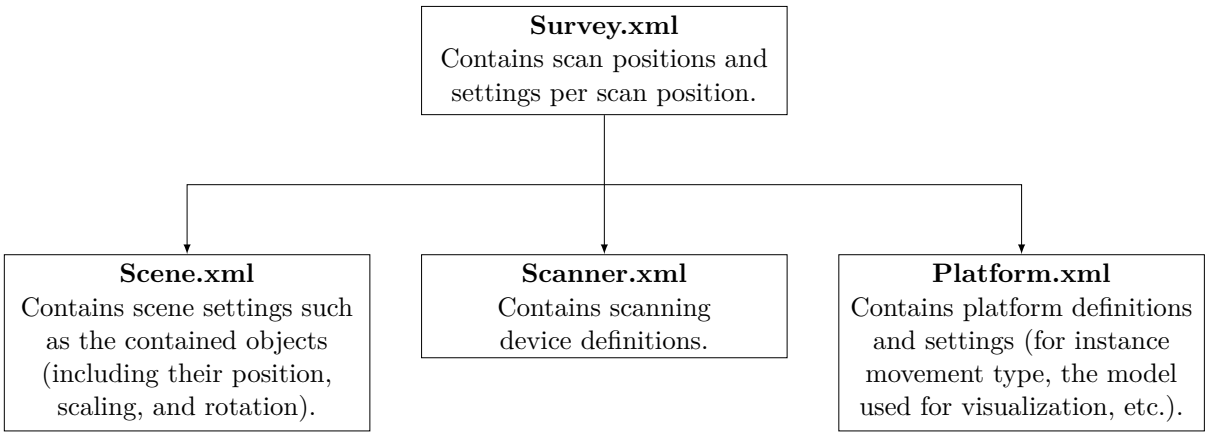


Figure 3.5: Illustration of the various XML files used in Helios. A *survey* operates a *scanner* on a *platform* within a *scene*. Adapted from [4].

simulation software Helios. In Blender, new scenes can be created by dragging and dropping objects in the workspace. It also enables users to import existing scenes that are not directly readable for Helios. We dedicate the whole chapter 5 to Blender2Helios.

Settings for our Simulated Riegl VZ-400 For the later application of Helios we defined our own Riegl VZ-400 laser scanner which is very similar to Helios' default settings for terrestrial laser scan simulations. We set the parameters as follows.

```
pulseFreq_hz="300000"
scanAngle_deg="50.0"
scanFreq_hz="120"
headRotatePerSec_deg="4.8"
headRotateStart_deg="0"
headRotateStop_deg="360"
```

This results in a field of view of 360° horizontally and 100° vertically with a resolution of 0.04° each as shown in equations 3.2 and 3.3. The calculations were done according to the documentation in [4].

$$resolution_{vertical} = 2 \cdot \frac{scanAngle\_deg \cdot scanFreq\_hz}{pulseFreq\_hz} = 2 \cdot \frac{50° \cdot 120\,\text{Hz}}{300000\,\text{Hz}} = 0.04° \tag{3.2}$$

$$resolution_{horizontal} = \frac{headRotatePerSec\_deg}{pulseFreq\_hz} = \frac{4.8°/\text{s}}{120\,\text{Hz}} = 0.04° \tag{3.3}$$

These settings match our real scanner's configuration. As previously mentioned, a resolution of 0.04° corresponds to a point spacing of $\tan(0.04°) \cdot 100\,\text{m} \approx 7\,\text{cm}$ at a distance of 100 meters.

3.4 Blender 2.8

Blender (`https://www.blender.org`) is an open-source 3D graphics software toolset first released in 1995. It is written in C, C++, and Python. With Blender, users can not only create complex 3D shapes and static 3D scenes but also animated 3D films. Many features like sculpting and soft body/fluid/particle simulations are supported. Another important capability is the scripting API. One can automate different tasks in Blender by executing corresponding Python commands. The API is fully integrated into Blender's GUI meaning that for most elements of the interface the associated API calls are displayed. We used Blender 2.81 throughout this thesis. Version 2.80 which introduced a whole new set of commands showed some bugs in the scripting engine.

Figure 3.6 presents a screenshot of Blender's main *Layout* tab. The main part of the window displays the 3D scenery with different objects. In the illustrated default scene only a cube, a camera, and a light source is included. On the right side of the screen all objects are hierarchically listed. Different objects can be grouped by putting them into the same collection which is named *Collection* here. The translation, rotation, and many other attributes of selected items can be seen and altered in the area in the lower right corner. The screenshot only serves as an overview. Due to the high number of features, Blender's interface is split into many different tabs that are completely customizable.

For a deeper introduction to Blender 2.8 readers are recommended to consult [19].

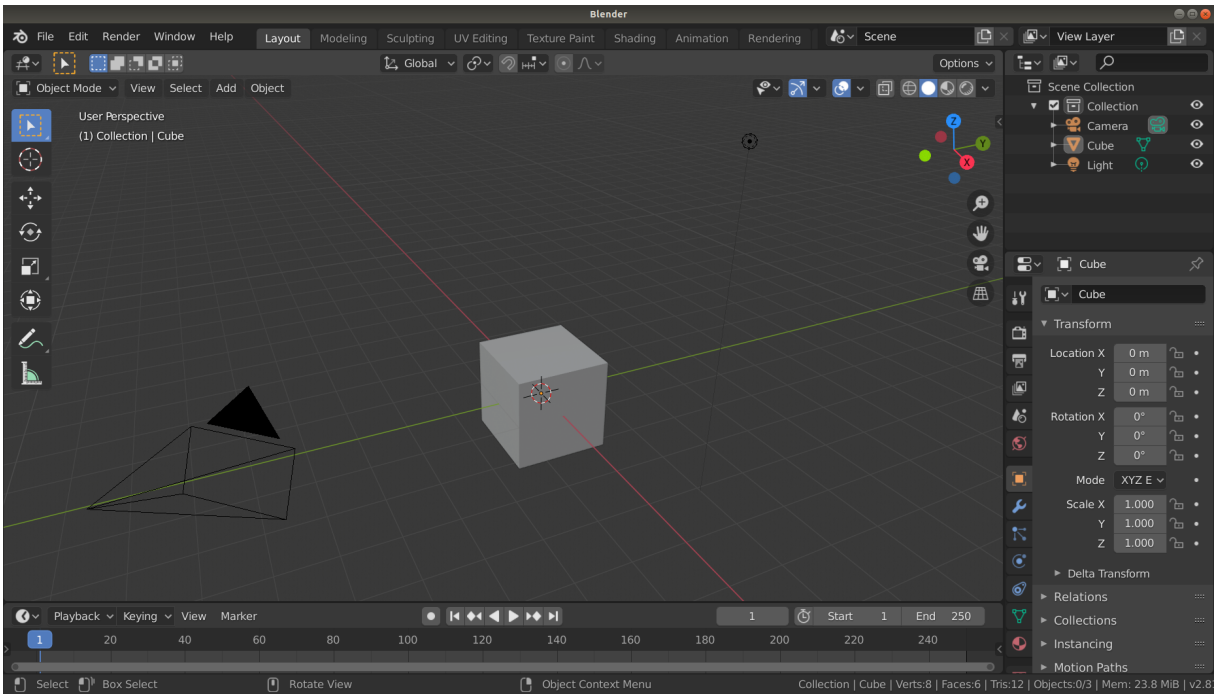


Figure 3.6: Screenshot of Blender's user interface currently showing the default scene with a cube, a camera, and a light.

Chapter 4

Datasets

Many benchmark datasets are available for 2D image classification. These are for instance *ImageNet* [31, 95] and *Pascal VOC* [37, 38] for rgb images and *SUN RGB-D* [101] for rgb-d data.[44] A huge amount of prelabeled training instances makes it possible to successfully apply modern machine learning algorithms. Often, the classifiers achieve accuracies far over 95%.

For 3D point clouds, however, less training data is available. Even well-known datasets often consist of only a few million points whereas one of our single laser scans already contains over ten million points. Compared to terrestrial laser scans most datasets like the *Oakland* [77], the *Sydney Urban Objects* [28], and the *IQmulus & TerraMobilita Contest* [109] datasets were created with mobile laser scanners. By design these are sparse compared to terrestrial laser scans. Instead of multiple minutes per turn around the vertical axis, the mobile scanners turn much faster. This allows to create a continuous but less dense point cloud during movement.

While the number of 3D points is still comparable to the number of pixels in 2D datasets, these datasets generally only consist of a small number of individual scans. This results in similar computation times while only learning on a few training examples, allowing over-fitting on the seen object instances.

The next sections in this chapter describe the datasets used in this thesis and their preprocessing. We also present scenes we scanned ourselves. These include real scans taken in Würzburg and simulated scans. The latter were facilitated by our new open-source Blender add-on Blender2Helios.

4.1 Oakland

The Oakland dataset is available for free for research purposes at `http://www.cs.cmu.edu/~vmr/datasets/oakland_3d/cvpr09/doc` and was first used in [77]. It consists of labeled 3D point clouds recorded in urban environment around the CMU campus in Oakland, Pittsburgh, PA. The data was collected from a moving car, the *NavLab11*. Side looking 2D Sick LMS laser scanners were mounted to create point clouds in a push broom fashion.

A preprocessed version of the complete dataset consisting of 1.6 million 3D points is available where splitting in training, testing, and validation data was already done. Furthermore, the 44 semantic classes were merged into five remaining ones as shown in table 4.1.

Table 4.1: Semantic classes in the preprocessed Oakland dataset.

Label			
Our	**Original**	**Class Name**	**Examples**
1	1004	Scatter/Misc	Foliage, Shrub
2	1100	Wire	Wire bundle, Isolated wire
3	1103	Pole	Post, Trunk
4	1200	Load/Bearing	Ground, Trail, Walkway, Grass
5	1400	Facade	Wall, Column
-	-	Removed	Bench, Garbage, Chimney, Human, Vehicle

The class distribution can be found in table 4.2. As the classifiers were tuned using another – the Semantic3D – dataset, we did not use the validation data given. We just used the training data to train the classifiers and the much bigger testing portion to test them.

Table 4.2: Class distribution in the Oakland dataset. Shown for the provided preprocessed data (table 4.1).

	Label					
Subset	**Scatter/Misc**	**Wire**	**Pole**	**Load/Bearing**	**Facade**	**Total**
Training	14,441	2,571	1,086	14,121	4,713	36,932
Validation	8,485	899	1,441	67,419	13,271	91,515
Testing	267,325	3,794	7,933	934,146	111,112	1,324,310

4.2 Paris-rue-Madame

The dataset fully called *Paris-rue-Madame database: MINES ParisTech 3D mobile laser scanner dataset from Madame street in Paris* is available at `http://www.cmm.mines-paristech.fr/~serna/rueMadameDataset.html`. It is under copyright of *MINES ParisTech* and was released under the *Creative Commons Attribution Non-Commercial No Derivatives (CC-BY-NC-ND-3.0)* license. To follow the dataset's conditions of use, we express that "MINES ParisTech created this special set of 3D MLS data for the purpose of detection-segmentation-classification research activities, but does not endorse the way they are used in this project or the conclusions put forward".[99]

The dataset consists of two laser scans containing exactly 10 million points each. Not only a semantic classification is given per point, but also an indication to what exact object each point belongs to. 624 objects were annotated and categorized in 26 classes.

The class distributions in the scans are shown in table 4.3. Due to the lack of more scans, only one scan (*scan 1_2*) was used for training while the other one (*scan 1_3*) was kept for testing. As many classes do not or only rarely occur in both scans, we additionally eliminated very small classes. The remaining are shown in table 4.4.

Table 4.3: Class distribution in the Paris-rue-Madame dataset.

		Number of Points	
Label	**Class Name**	**Training (1_2)**	**Testing (1_3)**
0	Background	3	11
1	Facade	4,769,417	5,209,018
2	Ground	4,333,059	3,691,236
4	Cars	790,822	1,044,561
7	Light poles	0	2,610
9	Pedestrians	3,656	6,392
10	Motorcycles	81,745	17,122
14	Traffic signs	11,463	4,017
15	Trash can	2,542	2,144
19	Wall Light	3,030	2,356
20	Balcony Plant	983	791
21	Parking meter	111	2,374
22	Fast pedestrian	1,915	7,515
23	Wall Sign	384	1,246
24	Pedestrian + something	491	0
25	Noise	379	5,173
26	Pot plant	0	3,434
	Total	10 mio.	10 mio.

4.3 Semantic3D

The highest quantity of training instances used in this thesis was provided by the *Semantic3D.net* [43] dataset. A static terrestrial laser scanner was used to create dense point clouds of outdoor scenes consisting of over four billion points. The scanned environments include churches, streets, railroad tracks, squares, villages, soccer fields, and castles in Central Europe.

Approximately one half of the points is published with class labels as training data. With the rest of the data, two public challenges are called out at `http://www.semantic3d.net`. In both, participants have to label point clouds and submit the results. The main challenge (*semantic-8*) contains over 2.3 billion unlabeled points in 15 scans. In comparison, one has to label approximately 80 million points partitioned in four scans in the smaller *reduced-8* challenge. Due to time constraints we only took part in the reduced-8 challenge.

Eight semantic classes were assigned by manual labeling: *man-made terrain*, *natural terrain*, *high vegetation*, *low vegetation*, *buildings*, *hard scape*, *scanning artefacts*, and *cars*. Table 4.5 pictures the distribution of class labels for the reduced-8 data.

Annotation in 3D is much more difficult than in 2D. For this dataset it was done using two different strategies[43]:

Table 4.4: Paris-rue-Madame's class distribution after preprocessing.

Label			Number of Points	
Our	**Orig**	**Class Name**	**scan1 (1_2)**	**scan2 (1_3)**
1	1	Facade	4,769,417	5,209,018
2	2	Ground	4,333,059	3,691,236
3	4	Cars	790,822	1,044,561
4	9	Pedestrians	3,656	6,392
5	10	Motorcycles	81,745	17,122
6	14	Traffic signs	11,463	4,017
Total			9,990,162	9,972,346

Table 4.5: Class distribution in the Semantic3D dataset (reduced-8).

		Number of Points	
Label	**Class Name**	**Training**	**Testing**
-	Unlabeled	156,046,453	11,607,777
1	Man-made Terrain	796,491,240	14,317,509
2	Natural Terrain	480,979,227	10,694,746
3	High Vegetation	135,634,808	4,675,871
4	Low Vegetation	99,971,504	3,021,894
5	Buildings	285,746,986	30,911,206
6	Hard scape	83,466,776	2,384,545
7	Artefacts	51,979,924	233,839
8	Cars	15,609,275	851,942
Total		2,105,926,193	78,699,329

Annotation in 3D Parts of the dataset were labeled by student assistants at the *ETH Zurich*. A 3D iterative filtering was performed by fitting simple models to a couple of manually selected points per object. Model outliers were then removed. This procedure was applied repeatedly until only inliers were left.

Annotation in 2D This procedure was used for outsourcing the annotation task. The user rotates a point cloud and fixes a 2D view. In this view he draws a closed polygon around the object. Only the data within this polygon is used in the next iteration where a new polygon is fitted. These steps are repeated until only class inliers are left. The tool *CloudCompare* [42] was used to apply this procedure.

4.4 Sim2Real

Instead of testing the classifiers only on already existing point clouds, we decided to test the performance on our own data as well. In this thesis we present one scene of a parking lot near the Department of Computer Science at the University of Würzburg. To determine the applicability of training classifiers on easy to generate, artificial scenes and using them for real world applications, we used two scenes. The first one was created in Blender using freely available object models. This one is used to train the classifier as it is much easier to get prelabeled point clouds of artificial scenes. The second one was taken at a real parking lot. It is used for testing.

Simulated Parking Lot Scan To have artificial training data available, we created our own parking lot scene in Blender. This allowed us to easily generate labeled point clouds by using Blender2Helios and Helios. Only freely available object models from *3D Warehouse* (`https://3dwarehouse.sketchup.com`) were used to coarsely remodel the real parking lot shown in the next paragraph. Paved/asphaltic ground was modeled by just a flat plane while a Blender modifier was used to represent grass. By applying a *Displacement Modifier* in Z direction combined with a *Voronoi texture* (`https://docs.blender.org/manual/en/latest/render/shader_nodes/textures/voronoi.html`), small spikes on a flat plane were created. We expected them to generate very similar points in the simulation as sampling real grass. As the classifiers showed that they learned the different ground types, we did not put more effort in a finer remodeling of the different types of ground. A big difference between artificial and real word scene is therefore still existent. This enforces the classifiers to generalize. One should be able to reproduce our results with his or her own 3D scenes without trying to build an exact copy of the testing data.

The created Blender scene can be seen from bird's eye view in the next chapter in figure 5.1.

Real Parking Lot Scan Scans of the real parking lot were taken during a normal working day. We used the previously introduced Riegl VZ-400 with the settings already shown in section 2.1.4. Naturally, these scans mainly contain paved ground and cars but also vegetation, natural ground, and buildings in the background. Figure 4.1 illustrates the on-site scenery.

To be able to use this data for testing, each point had to be assigned a class label. These labels were later used to calculate a classifier's accuracy. We decided for a division into six classes as shown in table 4.6.

Depending on the application, other class labels are reasonable. For instance, we decided stairs to be treated like other normal objects. This can be useful on a robot that can not pass stairs and where distinguishing between other items is not needed. In other scenarios they can for instance also be labeled as *man-made ground*. Similar decisions had to be made for other scenes we tested. How does somebody want to classify a balcony? Does it belong to a dedicated *balcony* class or does it belong to *facades*? Is the floor of a balcony *man-made ground*? Are the hand railings also part of the *balcony/facade* class or do they belong to *railings*? As every reader has a different focus, we decided to only print the results of this one scene. With Blender2Helios everybody is invited to create labeled point clouds of his or her own scenes and to apply the described learning algorithms.

Figure 4.1: Photograph of the parking lot where our scans were taken.

CloudCompare (`https://www.danielgm.net/cc`) was used for labeling the point cloud of the real parking lot. The cloud was first coarsely sliced using a polygon selection in 2D from top view. Then, these partitions were cleaned by iteratively applying polygon selections from different angles before they were exported. Labeling of the real point cloud and the object classification in Blender were done by the same person to reduce errors introduced by human aspects. At the time the scan was taken a construction side was nearby. To avoid any influence on the accuracy by unusual objects like a crane, the corresponding points were removed.

Table 4.6: Semantic classes in our Sim2Real dataset. The dataset is split into training data created with *Helios* and real testing data.

			Number of Points	
Label	**Class Name**	**Examples**	**Blender**	**Real**
1	Man-made ground	Paved ground, Asphalt	1,120,044	6,381,848
2	Natural ground	Grass, Grassland	128,813	209,347
3	Vegetation	Bushes, Trees	204,011	1,823,701
4	Object	Lamps, Poles, Railings, Stairs	16,900	64,575
5	Facade	Walls, Buildings	280,063	754,867
6	Car	All vehicles	161,480	1,478,190
		Total	1,911,311	10,712,528

Chapter 5

Blender2Helios - The Blender LiDAR Add-on

Blender2Helios is a Blender 2.8 add-on that was developed by the author during this research. It builds an interface between the common 3D creation suite Blender and the powerful LiDAR operations simulator Helios introduced in chapter 3. The add-on is released under GNU GPLv3. Code and documentation is available at `https://github.com/neumicha/Blender2Helios`.

Before simulating a laser scanning survey in Helios the scene has to be setup. Up until now, users had to write complex XML files on their own, describing translation and rotation of the individual 3D objects in the scene. While this is possible for small and undetailed artificial scenes, a more sophisticated way was desired to create very detailed, realistic scenes.

This is the benefit of our tool. It provides the possibility to build various scenes without much effort or even use existing ones. These scenes are then converted to Helios compatible XML files to be able to perform laser scan surveys subsequently. Also, semantic labels can be assigned easily by the collections feature in Blender.

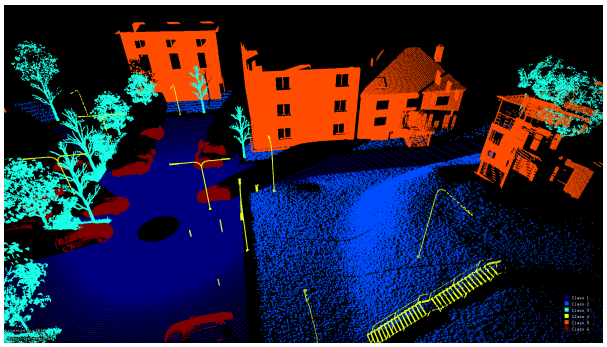
With our software we want to facilitate the creation of 3D point clouds in simulated scenes. Many realistic urban or natural environments, for instance created for games with the Unreal Engine, exist and can now be converted to semantically labeled point clouds.

Figure 5.1 shows the input and output of this toolchain on the example of a parking lot near the computer science building at the University of Würzburg. The scene includes details like street lamps, trees with and without leaves and even small poles and handrails. In section 7.3 we additionally show that classifiers trained on this artificial data can achieve high accuracies on real world scans.

Example of a *Scene.xml* file with two objects (denoted as parts) created by Blender2Helios. Abbreviations are indicated by "...".

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <scene id="ParkingLot" name="ParkingLot">
    <sunDir x="0" y="1" z="-1" />
    <skybox azimuth_deg="275" texturesFolder=".../sky6_1024" />
    <part>
      <filter type="objloader">
        <param type="string" key="filepath"
            value=".../Cars/Car-221-538-153.obj" />
        <param type="boolean" key="castShadows" value="true" />
        <param type="boolean" key="receiveShadows" value="true" />
        <param type="boolean" key="recomputeVertexNormals"
            value="true" />
      </filter>
      <filter type="rotate">
        <param type="rotation" key="rotation">
          <rot axis="pitch" angle_deg="0.0" />
          <rot axis="roll" angle_deg="0.0" />
          <rot axis="yaw" angle_deg="-173.0" />
        </param>
      </filter>
      <filter type="translate">
        <param type="vec3" key="offset" value="-35.0;-27.5;0" />
      </filter>
      <filter type="scale">
        <param type="double" key="scale" value="1.0" />
      </filter>
    </part>
    <part>
      <filter type="objloader">
        <param type="string" key="filepath"
            value=".../Vegetation/Birch-1307-1125-1487.obj" />
        ...
      </filter>
      <filter type="rotate">...</filter>
      <filter type="translate">...</filter>
      <filter type="scale">...</filter>
    </part>
  </scene>
</document>
```

(a) Blender scene created with freely available 3D models.



(b) Corresponding 3D point cloud generated with a simulated Riegl VZ-400 in Helios. Colors respresend class labels.

Figure 5.1: Parking lot scene created in Blender and laser scanned with Helios.

5.1 Main Challenges

Two main challenges occurred during the implementation of Blender2Helios. First, when for instance thinking of a forest scene, the same tree model might be used multiple times but with different translation, rotation, and scale. To speed up the export process and to save disk space, it is reasonable to export the model only once. While the individual translation and rotation can be easily set by the generated XML file, applying the correct scaling is not that easy. As Helios only supports uniform scaling in all directions, there are situations where not all instances of a tree can be represented by one model file. This is the case when different scaling factors were applied to different dimensions of an object in Blender. To at least minimize the number of object exports, we implemented a heuristic to decide whether an object needs to be exported or if it can be replaced by another model.

For the heuristic, the subsequent two rules are checked.

1. The names of both objects in the scene are very similar, i.e. the leading parts of the names, until a probably existing period, are exactly the same. The period used as a delimiter is part of Blender's naming convention. When importing a model called *car* multiple times, the objects will be named *car*, *car.001*, *car.002*, et cetera by default.

2. The size after normalization of the scaling is identical. For a Blender object with outer dimensions $[d_x \;\; d_y \;\; d_z]^\intercal$ with applied scaling factors $[s_x \;\; s_y \;\; s_z]^\intercal$ the normalized outer dimensions are calculated as $\frac{[d_x \;\; d_y \;\; d_z]^\intercal}{s_x}$. This calculation brings the scaling in x direction to 1 which usually represents the original size.

These rules can be checked with low computational cost compared to a complex comparison of vertices and edges. Only if both conditions are satisfied, just one of the objects is exported. Although this heuristic may generate a small number of false positives, they can be easily bypassed by a clear naming convention for unambiguous object names on the user's side to never erroneously satisfy the first rule.

The second challenge was the different representation of rotations in Blender and Helios. All the rotations of objects had to be converted to Tait-Bryan angles. *Roll* (ϕ), *pitch* (θ), and *yaw* (ψ) for intrinsic rotations around the three object axes had to be calculated. For this we used the representation of quaternions of Blender ($\boldsymbol{q} = [w \ x \ y \ z]^\intercal$) and did a conversion according to [8]. In combination with the axis settings during the model export and the order of rotations in Helios this yields the solution shown in equation 5.1. Note that multiple solutions exist as different orders of the rotation axes are imaginable. This order of rotations, however, is the standard provided with examples of Helios.

$$\begin{aligned} \phi &= \arcsin(2(wy - zx)) \\ \theta &= \operatorname{arctan2}(2(wx + yz), 1 - 2(x^2 + y^2)) \\ \psi &= \operatorname{arctan2}(2(wz + xy), 1 - 2(y^2 + z^2)) \end{aligned} \tag{5.1}$$

Part of the generated *Scene.xml* describing the rotation of an object:

```
...
<param type="rotation" key="rotation">
  <rot axis="pitch" angle_deg="rad2deg(θ)"/>
  <rot axis="roll" angle_deg="rad2deg(ϕ)"/>
  <rot axis="yaw" angle_deg="rad2deg(ψ)"/>
</param>
...
```

5.2 How to Use

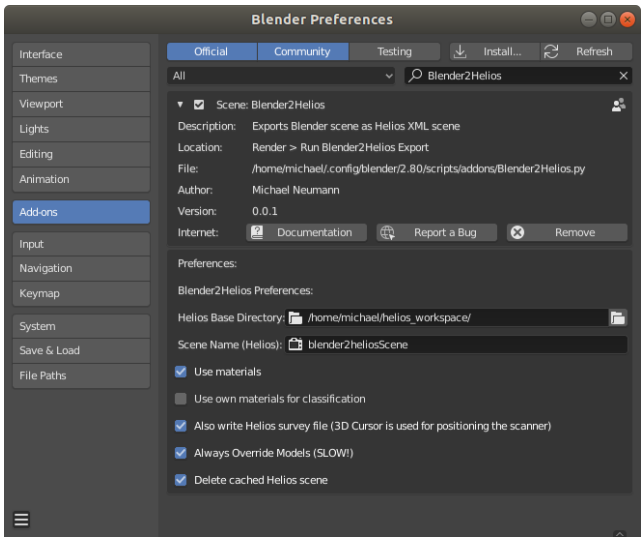
After installing and enabling the add-on, a number of parameters can be changed in the preferences dialogue. Figure 5.2 shows the corresponding user interface containing the subsequent features:

Helios Base Directory Points to the directory containing the folders *assets* and *data* of Helios. It will be used as target path for the later scene export. The .obj models of the scene will be stored in *data/sceneparts* within this folder.

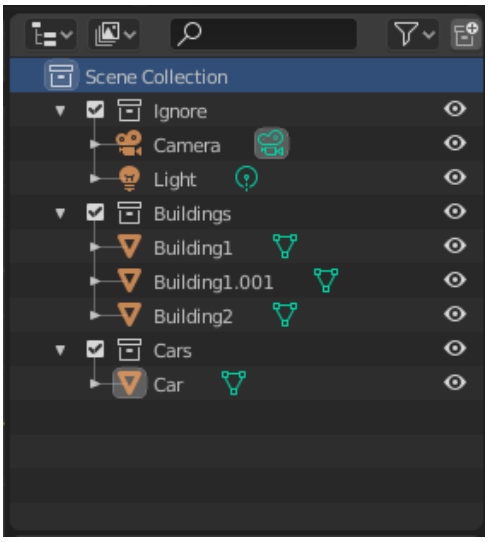
Scene Name This string is used as name for the exported Helios scene.

Use materials Defines if the materials of Blender objects are also exported. If disabled (and the next flag for own materials is also disabled), the objects will just show up black in Helios. However, this does not affect the spatial data of the simulated laser scans.

Use own materials for classification When enabled, all materials will be replaced depending on the collection each object belongs to. This allows to create point clouds with semantic labels. The user has to provide a valid *materials.mtl* for Helios manually. The way this is done is shown after this parameter description.



(a) Preference window in Blender for the Blender2Helios add-on. It allows to set the main parameters for scene exports.



(b) Typical collection setup in Blender to be used with Blender2Helios.

Figure 5.2: Blender2Helios's preferences and scene preparation in Blender.

Also write Helios survey file Exporting also generates a survey file for Helios. The location of Blender's 3D cursor is used as location for the laser scanner. A default 360° setup with the Riegl VZ-400 is used. The generated survey file may be altered by the user subsequently depending on the use case.

Always override models The model files are stored according to the naming convention *data/sceneparts/[COLLECTION]/[OBJNAME]-[SIZE].obj*. *[OBJNAME]* is replaced by the name of an object and *[COLLECTION]* indicates the Blender collection it belongs to. As previously discussed, a faster export is possible by consolidating objects that occur more than once in the scene. Different translations, rotations, and scalings are possible by exporting the object once and providing different settings in the *Scene.xml* file. The naming convention helps to accomplish this. *[OBJNAME]* is cut at the first period which is typically introduced by Blender when duplicating an object within the scene. *[SIZE]* represents the normalized object size. This convention enables easy caching of objects that occur multiple times with different scaling and that were already exported before. As long as users don't work with extremely big scenes, leaving this setting enabled is reasonable. If this setting is disabled (faster) obsolete exported objects have to be manually removed from the *sceneparts*, otherwise the export will not replace existing files.

Delete cached Helios scene If a scene with the given scene name was already opened in Helios, a cache was created. It contains Helios's internal data structure for shorter loading times. After exporting a new scene one usually wants Helios to open this newly created one. If this settings is enabled, Helios's tree cache is deleted during the scene export procedure.

To group objects by their semantic labels and use these in the later laser scanning survey in Helios, we exploit the feature of setting different materials for objects. Blender2Helios expects all objects within collections. A typical setup of collections and objects used in Blender2Helios is also shown in figure 5.2. The collection with the name *ignore* is ignored and can for instance be used for cameras, lights, or other objects that are unwanted in the laser scan. For all other collections the assigned objects are exported as .obj [16] files to a folder with the name of this collection. When the flag *Use own materials for classification* is set, each exported object file is modified to use a user-defined material instead of the original materials. The following statements are prepended at every .obj file.

Header of an exported .obj file for a building with *Use own materials for classification* enabled.

```
mtllib ../materials.mtl
usemtl Buildings
...
```

To bring big scenes with unsorted objects into a suited format of collections without too much manual effort, a scene conversion script is also provided in the Github repository. It moves all items into collections according to some user-defined rules. This is done by pattern-matching of each object's name. The script was tested with a big Unreal Engine scene with around 3,000 objects. It was able to classify over 95% of the objects by rules that were created within minutes. Additionally, a user has to define the mapping of collections to class labels. This is done by a *materials.mtl* file which might contain the subsequent lines.

Content of *materials.mtl* (partially shown):

```
newmtl Buildings
Ka 0 0 1
helios_classification 1

newmtl Cars
Ka 1 0 0
helios_classification 2
...
```

This results in a colored Helios scene with buildings being blue (*Ka 0 0 1*) and cars being red (*Ka 1 0 0*). The proprietary *helios_classification* property sets the later class label for each material. Simulated laser scans will include this value when a 3D point was created by the laser hitting the corresponding material. [90] provides more details about the material file format.

5.3 Implementation

Blender2Helios is implemented as a real add-on for the Blender software. Due to the overhead in code for complying with the add-on conventions, we decided to print its main functionality in form of pseudocode in this thesis in algorithms 3 and 4. This will give an idea how the extension works and enables readers to easily understand and modify the code. The small helper functions are briefly described in appendix A. The newest operational code is always released at `https://github.com/neumicha/Blender2Helios`.

Algorithm 3 Blender2Helios - Schema of exporting a scene (Part 1)

```
1: From os.path import join
2:
3:       // Defaults shown. May be changed by the user in the add-on's preferences
4: HELIOS_BASE_DIR ← "/home/USERNAME/"
5: SCENE_NAME ← "blender2heliosScene"
6: ALSO_WRITE_SURVEY_FILE ← true
7: ALWAYS_OVERRIDE_MODELS ← true
8: DELETE_CACHED_SCENE ← true
9: USE_MATERIALS ← true
10: USE_OWN_MATERIALS ← false
11:
12: if DELETE_CACHED_SCENE then
13:     Delete join(HELIOS_BASE_DIR,"data/scenes",SCENE_NAME+".scene")
14: end if
15: fScene ← Open join(HELIOS_BASE_DIR,"data/scenes",SCENE_NAME+".xml")
16: fScene.write(xmlSceneHead())
17: fScene.write(buildSceneParts())      // The main work is done here, see algorithm 4
18: fScene.write(xmlSceneFoot())
19: fScene.close()
20: if ALSO_WRITE_SURVEY_FILE then
21:     fSurvey ← Open join(HELIOS_BASE_DIR,"data/surveys",
            SCENE_NAME+".xml")
22:     fSurvey.write(xmlSurvey())
23:     fSurvey.close()
24: end if
```

Algorithm 4 Blender2Helios - Schema of exporting a scene (Part 2)

```
25: function BuildSceneParts( )
26:     out ← ""
27:     for all c ∈ Collections with c.name ≠ "ignore" do
28:         cName ← c.name.until(".")      // Cut name at first period
29:         for all o ∈ c.objects do
30:             oName ← o.name.until(".")      // Cut name at first period
31:             objFileExtension ← toText(scale2Original(o))
32:             objFile ← join(HELIOS_BASE_DIR,"data/sceneparts",cName,
                    oName+"-"+objFileExtension+".obj")
33:             scale ← o.scale[0]      // Later export it with a scaling of 1 in first dimension
34:             o.rotationMode ← "QUATERNION"      // Needed to access the data later
35:             if ALWAYS_OVERRIDE_MODELS ∨ ¬exists(objFile) then
36:                 Backup rotation and translation of o
37:                 o.location ← 0
38:                 o.scale ← o.scale / scale
39:                 Select only object o
40:                 Export selected object to objFile [including materials
                        ⟺ (USE_MATERIALS ∧ ¬USE_OWN_MATERIALS)]
41:                 Restore rotation and translation of o
42:                 if USE_OWN_MATERIALS then
43:                     Prepend material with name cName to objFile
44:                 end if
45:                 out ← out + object2XML(cName, oName+"-"+objFileExtension+".obj",
                        o.translation, quaternion2RPY(o.rotation_quaternion), scale)
46:             end if
47:         end for
48:     end for
49:     return out
50: end function
```

Part II

Semantic Classification

Chapter 6

Implementation and Tuning of the Classifiers

This chapter mainly describes our more classical approach by calculating local features of each 3D point and using common classification algorithms afterwards. Compared to the original implementation in [115], we do feature calculation on multiple differently reduced scans. For classification we tune the parameters of a random forest and two SVMs. Only the last subsection, 6.4, describes an alternative way by using a neural network that is very similar to a CNN.

The process of parameter selection was always performed on the Semantic3D dataset. For this, the prelabeled training part was split into two parts. One part, containing twelve of the fifteen scans, was used for training while the remaining three scans (*untermaederbrunnen1*, *untermaederbrunnen3*, and *sg28_4*) were used for validation. The resulting parameters were then used to train and test the classifiers on all the introduced datasets including the Semantic3D reduced-8 challenge. Chapter 7 shows their comparison.

For the classical approach of calculating local features in the 3D point cloud and applying general classifiers, the order of parameter tuning had to be decided. When evaluating different feature sets and neighborhood parameters for feature calculation, one already needs a classifier to assess their impact. However, when tuning parameters of the classifier, one already needs expressive features for training. We decided to first optimize the feature calculation on the basis of a standard random forest setup with 100 trees and a small tree depth of four. Similar configurations were already used in literature [104, 114, 115] and proved to be sufficient to detect changes generated by different parameters in the feature calculation.

To allow the testing of many configurations, features were calculated on only a reduced number of points. An octree-based reduction with a voxel size of 10 cm was done and features were only calculated for the resulting points. Afterwards, the data was subsampled to bring the classes to a uniform distribution with approximately 63,000 points per class. However, for the calculation of feature values all points were used.

Appendix B shows measures of random forest classifiers with a different number of trees and with different random seeds for data subsampling and the forest creation. Features of one reduction scale served as input. One sees that the results are almost stable, i.e. within 2% for the tested seeds, and that 100 trees are sufficient for representative results.

6.1 Finding the Optimal Neighborhood Size

To calculate local features, a number of neighbors must be taken into account. For this set of points the principal components are distinguished using PCA. Typical approaches often use a sphere with a fixed radius [65] or a cylindrical neighborhood definition [39]. Another approach is simply using a fixed number of k nearest neighbors [62] to define the neighborhood.

In [115] a state-of-the-art approach which is based on dimensionality features [30] is presented. Instead of defining a fixed neighborhood size, different ways to find an *optimal neighborhood size* k_{Opt} per point are compared. The best results were achieved by choosing a neighborhood size that minimizes the eigenentropy. Compared to the common default of using the nearest $k = 100$ neighbors for feature calculation, this method yielded 2% to 14% higher overall accuracy scores depending on the dataset.

Eigenentropy E_λ is given by the Shannon entropy of eigenvalues λ_i. Its calculation for the three-dimensional case is depicted in equation 6.1. To reduce the impact of the local point density, normalized eigenvalues Λ_i as defined in equation 6.2 are used in our work.

$$E_\lambda = -\Lambda_1 \ln(\Lambda_1) - \Lambda_2 \ln(\Lambda_2) - \Lambda_3 \ln(\Lambda_3) \tag{6.1}$$

$$\Lambda_i = \frac{\lambda_i}{\sum_{j=1}^{3} \lambda_j} \tag{6.2}$$

The three normalized eigenvalues are constrained by $\sum_{i=1}^{3} \Lambda_i = 1$. It was therefore possible to create a heatmap of the eigenentropy in 3D space. Figure 6.1 shows this. When searching for the optimal neighborhood size, one minimizes the eigenentropy, so blue regions in the heatmap are favored. One can deduce from the figure that one-dimensional structures are priviledged. The function's maximum of $-\ln(1/3) \approx 1.0986$ is reached by neighborhoods with equally distributed points along all three axes, resulting in $\lambda_1 = \lambda_2 = \lambda_3$ and, accordingly, $\Lambda_1 = \Lambda_2 = \Lambda_3 = 1/3$.

The optimal number of neighbors k_{Opt} is found by repeatedly calculating the eigenentropy of neighborhoods with a varying number of neighbors $k \in \{k_{Min}, k_{Min} + k_\Delta, \ldots, k_{Max}\}$. k_{Min}, k_{Max}, and k_Δ are parameters that need to be set by the user. Common values are $k_{Min} = 10$, $k_{Max} = 100$, and $k_\Delta = 1$. This includes the often fixed value $k = 100$ and limits the number of times the eigenvalues have to be calculated per point. By increasing the step width k_Δ, one can impact the run time for distinguishing the optimal neighborhood sizes by a constant factor.

Figure 6.2 illustrates the time needed to query the neighborhoods of 1 million points. One can see that bigger neighborhoods with 500 or even 1,000 neighbors take longer to find and to perform the PCA on. Especially trying to find optimal neighborhood sizes within big ranges is very time consuming.

Table 6.1 shows the overall accuracy and class-wise F_1 scores when using the optimal neighborhood definition. Using between 500 and 1,000 neighbors for a bigger area of influence showed the best results. However, using that many neighbors for feature calculation is very time consuming as previously presented in figure 6.2.

We therefore decided to follow an approximation approach similar to [45]. Instead of calculating features on a large number of neighbors, a smaller number of neighbors is used within a reduced representation of the point cloud. This allows a fast calculation while still taking a wide spatial area into account. Octree-based reductions (see chapter 2.2.1) are used to create the less dense point clouds.

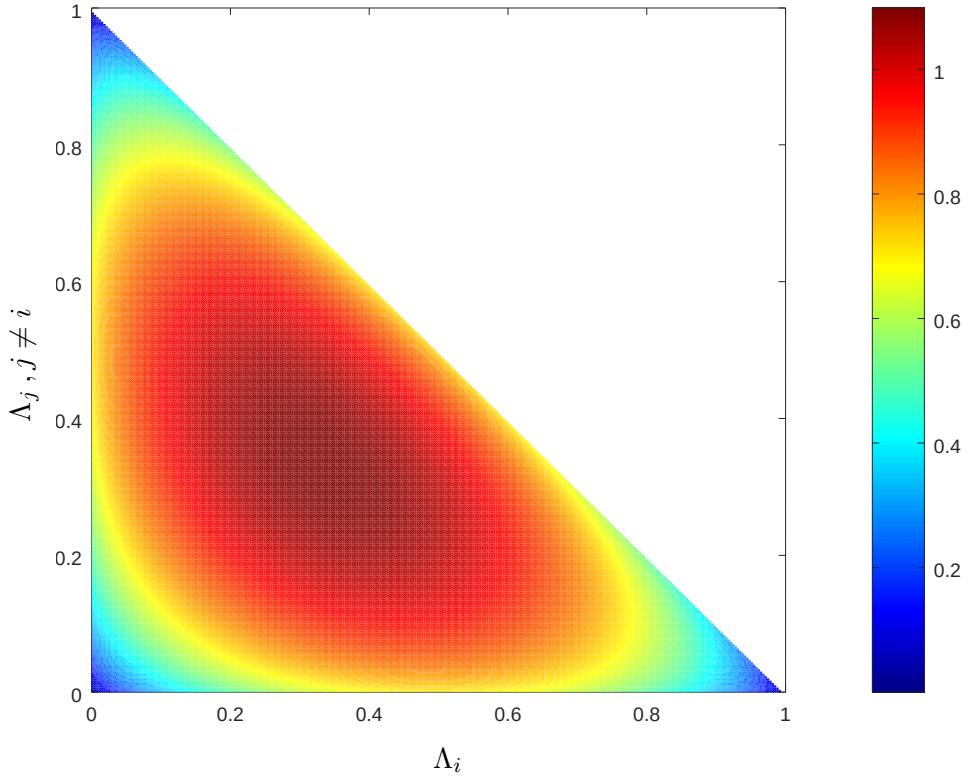


Figure 6.1: Illustration of eigenentropy in 3D space. The axes describe two normalized eigenvalues $\Lambda_i, \Lambda_j, i \neq j$ while the third eigenvalue can be calculated accordingly. Colors in the heatmap indicate the corresponding value of eigenentropy. Lower values (blue) are favored when searching for optimal neighborhoods.

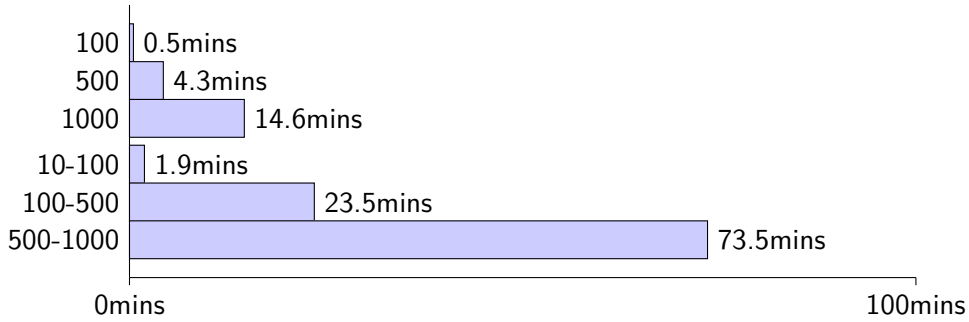


Figure 6.2: Time needed for calculating features with different neighborhood sizes. Calculations were done for one million 3D points in a typical outdoor scan. The first three measurements were taken with a fixed neighborhood size (100, 500, 1000) whereas for the last three ones our optimal neighborhood definition was used.

Table 6.1: Impact of varying ranges for finding the optimal neighborhood size. The F_1 scores of our random forest classifier are shown.

	Terrain		Vegetation		Build-	Hard	Arte-		Accu-
N. size	**Man-m.**	**Nat.**	**High**	**Low**	**ings**	**Scape**	**facts**	**Cars**	**racy**
10-100	**0.916**	0.359	0.749	**0.307**	0.583	0.157	0.227	0.123	64.26%
10-500	0.913	**0.368**	0.749	0.300	0.608	0.153	0.225	0.126	64.49%
100-500	0.822	0.298	**0.789**	0.177	**0.729**	0.139	0.119	0.123	64.58%
500-1000	0.904	0.249	0.783	0.229	0.715	**0.213**	0.355	0.122	**68.36%**
1000-5000	0.880	0.107	0.752	0.289	0.688	0.024	**0.410**	**0.168**	66.46%
10-5000	0.893	0.278	0.749	0.300	0.621	0.144	0.243	0.157	64.16%

6.2 Definition of Multiscale Features

[114, 115] also introduced a set of 21 features for every 3D point. [54, 70] describe the underlying principle of calculating invariant moments representing geometric properties as well as the 3D structure known from the covariance matrix. The definitions of the special geometric properties that were exploited are based on [71, 107].

We slightly changed the original feature definition of [115]. One reason was the aim to reduce the assumptions that were made. While the original features, for instance, include the absolute height of a 3D point, we decided to calculate a point's height with respect to the lowest point within a cylindrical neighborhood with a fixed radius. To some extend this allows to mix differently created 3D scans as the scanner height does not matter. Also some inaccuracies were found in [115]. One of the described features is the sum of eigenvalues given by $\Sigma_{\lambda,original} = \Lambda_1 + \Lambda_2 + \Lambda_3$. For the normalized values Λ_i, however, it always holds equation 6.3 and therefore $\Sigma_{\lambda,original} = 1$. As this feature does not store any information about the data, we used the sum of unnormalized eigenvalues λ_i to define a feature.

$$\Sigma_{\lambda,original} = \Lambda_1 + \Lambda_2 + \Lambda_3 = \frac{\lambda_1}{\sum_{i=1}^{3}\lambda_i} + \frac{\lambda_2}{\sum_{i=1}^{3}\lambda_i} + \frac{\lambda_3}{\sum_{i=1}^{3}\lambda_i} = \frac{\lambda_1+\lambda_2+\lambda_3}{\sum_{i=1}^{3}\lambda_i} = 1 \tag{6.3}$$

The 21-dimensional feature vector for each point $\boldsymbol{p_i}$ in a point cloud P is given by equations 6.4 to 6.7. λ_i and λ_i^{2D} denote the i-th eigenvalue of the 3D, respectively 2D, covariance matrices of the k neighboring points $\{\boldsymbol{q_1}, \boldsymbol{q_2}, \ldots, \boldsymbol{q_k}\}$ with mean $\boldsymbol{\mu} = \frac{1}{k}\sum_{1\leq j\leq k}\boldsymbol{q_j}$ around point $\boldsymbol{p_i}$. The neighbors are ordered by their distance to $\boldsymbol{p_i}$ ($||\boldsymbol{q_j} - \boldsymbol{p_i}|| \leq ||\boldsymbol{q_l} - \boldsymbol{p_i}||$ for $j \leq l$). Two-dimensional data is created by projecting all the points $\boldsymbol{p_i}$ to two-dimensional points $\boldsymbol{p_i^{2D}}$ on the ground plane by ignoring the vertical axis. The neighbors $\{\boldsymbol{q_1^{2D}}, \boldsymbol{q_2^{2D}}, \ldots, \boldsymbol{q_k^{2D}}\}$ are again ordered by ascending distance to the reference point. We further order the eigenvalues and the corresponding eigenvectors $\boldsymbol{e_i}$ in a way that $\lambda_1 \geq \lambda_2 \geq \lambda_3 \geq 0$ and $\lambda_1^{2D} \geq \lambda_2^{2D} \geq 0$ hold true. Λ_i denotes the normalized equivalents of λ_i again. Calculating normalized Λ_i^{2D} is not needed for this set of features as it does not impact the value of $Ratio_k^{2D}(\boldsymbol{p_i}, \boldsymbol{P})$. In all equations the subscript $_{vert}$ means the vertical dimension. This is often the third axis, but in case of 3DTK the second one.

For the cylindrical features we further let $\boldsymbol{q_j^{Cyl}}, 1 \leq j \leq k^{Cyl}$ be points within a cylindrical neighborhood with radius r around point $\boldsymbol{p_i}$. $\boldsymbol{\mu^{Cyl}} = \frac{1}{k^{Cyl}} \sum_{1 \leq j \leq k^{Cyl}} \boldsymbol{q_j^{Cyl}}$ denotes their mean.

$$\boxed{\boldsymbol{features(p_i, P)} = \begin{bmatrix} \boldsymbol{features_k^{3D}(p_i, P)} \\ \boldsymbol{features_k^{2D}(p_i, P)} \\ \boldsymbol{features_r^{Cyl}(p_i, P)} \end{bmatrix}} \tag{6.4}$$

$$\boldsymbol{features_k^{3D}(p_i, P)} = \begin{bmatrix} Linearity_k^{3D}(\boldsymbol{p_i}, P) \\ Planarity_k^{3D}(\boldsymbol{p_i}, P) \\ Scattering_k^{3D}(\boldsymbol{p_i}, P) \\ Omnivariance_k^{3D}(\boldsymbol{p_i}, P) \\ Anisotropy_k^{3D}(\boldsymbol{p_i}, P) \\ Eigenentropy_k^{3D}(\boldsymbol{p_i}, P) \\ SumOfEigenvalues_k^{3D}(\boldsymbol{p_i}, P) \\ ChangeOfCurvature_k^{3D}(\boldsymbol{p_i}, P) \\ Verticality_k^{3D}(\boldsymbol{p_i}, P) \\ LocalDensity_k^{3D}(\boldsymbol{p_i}, P) \\ Radius_k^{3D}(\boldsymbol{p_i}, P) \\ MaxHeightDifference_k^{3D}(\boldsymbol{p_i}, P) \\ HeightVariance_k^{3D}(\boldsymbol{p_i}, P) \end{bmatrix} = \begin{bmatrix} \frac{\Lambda_1 - \Lambda_2}{\Lambda_1} \\ \frac{\Lambda_2 - \Lambda_3}{\Lambda_1} \\ \frac{\Lambda_3}{\Lambda_1} \\ \sqrt[3]{\Lambda_1 \cdot \Lambda_2 \cdot \Lambda_3} \\ \frac{\Lambda_1 - \Lambda_3}{\Lambda_1} \\ -\sum_{i=1}^{3} \Lambda_i \cdot ln(\Lambda_i) \\ \sum_{i=1}^{3} \lambda_i \\ \Lambda_3 \\ 1 - |(\frac{\boldsymbol{e_3}}{||\boldsymbol{e_3}||})_{vert}| \\ \frac{k}{4/3 \cdot \pi \cdot Radius_k^{3D}(\boldsymbol{p_i}, P)^3} \\ ||\boldsymbol{q_k} - \boldsymbol{p_i}|| \\ \max_{1 \leq i \leq k}(q_{i,vert}) - \min_{1 \leq i \leq k}(q_{i,vert}) \\ \frac{1}{k} \sum_{i=1}^{k} (q_{i,vert} - \mu_{vert})^2 \end{bmatrix} \tag{6.5}$$

$$\boldsymbol{features_k^{2D}(p_i, P)} = \begin{bmatrix} Radius_k^{2D}(\boldsymbol{p_i}, P) \\ LocalDensity_k^{2D}(\boldsymbol{p_i}, P) \\ SumOfEigenvalues_k^{2D}(\boldsymbol{p_i}, P) \\ Ratio_k^{2D}(\boldsymbol{p_i}, P) \end{bmatrix} = \begin{bmatrix} \left|\left| \boldsymbol{q_k^{2D}} - \boldsymbol{p_i^{2D}} \right|\right| \\ \frac{k}{\pi \cdot Radius_k^{2D}(\boldsymbol{p_i}, P)^2} \\ \sum_{i=1}^{2} \lambda_i^{2D} \\ \frac{\lambda_1^{2D}}{\lambda_2^{2D}} \end{bmatrix} \tag{6.6}$$

$$\boldsymbol{features_r^{Cyl}(p_i, P)} = \begin{bmatrix} NrPoints_r^{Cyl}(\boldsymbol{p_i}, P) \\ MaxHeightDifference_r^{Cyl}(\boldsymbol{p_i}, P) \\ HeightVariance_r^{Cyl}(\boldsymbol{p_i}, P) \\ HeightAboveMin_r^{Cyl}(\boldsymbol{p_i}, P) \end{bmatrix} = \begin{bmatrix} k^{Cyl} \\ \max_{1 \leq i \leq k^{Cyl}}(q_{i,vert}^{Cyl}) - \min_{1 \leq i \leq k^{Cyl}}(q_{i,vert}^{Cyl}) \\ \frac{1}{k^{Cyl}} \sum_{i=1}^{k^{Cyl}} (q_{i,vert}^{Cyl} - \mu_{vert}^{Cyl})^2 \\ p_{i,vert} - \min_{1 \leq i \leq k^{Cyl}}(q_{i,vert}^{Cyl}) \end{bmatrix} \tag{6.7}$$

We also propose the feature calculation with different neighborhood sizes on differently reduced scans. This addition is illustrated in the next sections.

6.2.1 Implementation in 3DTK

The tool *scan2features* was implemented as part of 3DTK. It provides an easy way to calculate local features in point clouds. For faster execution it uses the multithreading library *OpenMP*. After compilation the tool can be found in the *bin* directory. It can be called, inter alia, with the subsequent options.

Excerpt of the command line parameters of *bin/scan2features*:

```
$ bin/scan2features --help

Input options:
-f [--format] arg (=uos)  Input: uos, uosc, xyz, xyzc, ply, ...

Algorithm options:
-a [--algorithm] arg (=0) Algorithm mode. (More may follow)
                           0 = Calc. features with kOpt
                            on whole scan (Weinmann 2014)
                             use: kMin, kDelta, kMax, cylRadii
                             set kMin=kMax for fixed kNN

Feature construction options:
-r [--reduce] arg (=10)    Reduction (cm) for reference points
-i [--ignoredClasses] arg  Class labels to ignore (only ref. pts)

-R [--kReductions] arg (=10) Reductions (cm) for 2D/3D features
-k [--kMins] arg (=10)       Minimum number of neighbors (kOpt)
-K [--kMaxs] arg (=100)      Maximum number of neighbors (kOpt)
-d [--kDeltas] arg (=1)      Step widths between kMins and kMaxs

--cylReductions arg (=10)    Reductions (cm) for cyl. features
-c [--cylRadii] arg (=25)    Radii (cm) of cylinders

Output options:
--csv         Output features as .csv file instead of .arff (Weka)

Example usage:
./bin/scan2features -r 0 --kReductions 2 20 80
   --kMins 10 10 10 --kMaxs 100 100 100 --kDeltas 1 1 1
   --cylReductions 2 20 80 --cylRadii 25 150 500 -f xyzc .
```

Scan2features runs with different input file types depending on the presence (*xyzc*, *uosc*) or absence (*xyz*, *uos*, etc.) of class labels. The standard algorithm calculates features for whole point clouds. The *--reduce* parameter can be used to determine the subset of points that features are calculated for. We call them *reference points*. Depending on the parameter's value, the point cloud is first reduced and the feature extraction is only done for the resulting points. This does not impact the features values themselves. Additionally, the *--ignoredClasses* feature allows to provide a list of class labels to ignore. Corresponding points still impact the feature values of other points but they are never used as reference points for feature calculation.

The feature construction options furthermore include *--kReductions*, *--kMins* , *--kMaxs*, and *--kDeltas* which highly impact the resulting feature values. All four parameters hold lists of the same length, here denoted as k_{len}. For each $i \in \{0, 1, \ldots, k_{len} - 1\}$ a reduced scan with a voxel size of $k_{Reductions}[i]$ cm is generated. Within this cloud, denoted as $P_{k_{Reductions}[i]}$, the 2D and 3D features of an optimal neighborhood between $k_{Mins}[i]$ and $k_{Maxs}[i]$ with a step size of $k_{Deltas}[i]$ are calculated. Choosing $k_{Mins}[i] = k_{Maxs}[i]$ results in the calculation based on a fixed number of neighbors. By setting *--cylReductions* and *--cylRadii* one defines equivalent scan reductions and radii used for the calculation of the cylindrical portion of the feature vector. The *--csv* flag can be used to write the features to csv files instead of arff files for Weka.

The shown example call calculates features for all (*-r 0*) input points of the xyzc scans (*-f xyzc*) within the current directory (.). The 2D and 3D features are calculated based on three reduced scans with voxel sizes of 2 cm, 20 cm, and 80 cm (*--kReductions 2 20 80*). On all levels a range of 10 to 100 neighbors is used to find the optimal neighborhood size (*--kMins 10 10 10 --kMaxs 100 100 100 --kDeltas 1 1 1*). The additional features based on cylinders are also created on the same reduced scans (*--cylReductions 2 20 80*) with radii of 25 cm, 150 cm, and 500 cm, respectively (*--cylRadii 25 150 500*).

Algorithm 5 shows a schema of the algorithm used in scan2features to calculate features for a whole point cloud. To reduce the number of neighborhood searches, the k_{Max} nearest neighbors are always queried once and used to find k_{Opt} instead of iteratively querying different neighborhood sizes. Scan2features outputs three files. The file *features.arff* holds the calculated features of one point per line. *features_points.3d* contains the reference points in the order they were processed. Due to multi-threading, the order of points may differ to the input. Additionally, the discovered optimal neighborhood sizes are stored in a file called *features_kOpt.arff*. This file can be used for further analysis or could – with some changes in the code – be used to speed up later feature calculations on the same scans. The latter may be useful for comparing different feature sets in combination with a constant optimal neighborhood definition.

6.2.2 Parameter Optimization

The best parameters for the approximate feature calculation had to be found. A number of different reduction parameters was tested with $10 \leq k_{Opt} \leq 100$. Class-wise F_1 scores and accuracies are printed in table 6.2. Precision and recall measures can be found in appendix C. One can see that different reductions allow different classes to be recognized better. For instance, scanning artefacts are identified with higher accuracy in a detailed scan and a spatial small neighborhood whereas buildings are recognized more often within a wider area in heavily reduced scans. Voxel sizes over 80 cm seem to negatively impact the accuracy.

Algorithm 5 The feature calculation algorithm used in *scan2features*

Input: Point cloud P, Reduction parameter *reduce*,
Lists $k_{Reductions}$, k_{Mins}, k_{Maxs}, k_{Deltas} of length k_{len},
Lists $cyl_{Reductions}$, cyl_{Radii} of length cyl_{len}
Output: Files: features.arff, features_kOpt.txt, features_points.3d

1: $reductions \leftarrow reduce \cup k_{Reductions} \cup cyl_{Reductions}$ // *Set of all reduction scales (no dublicates)*
2: **for all** $red \in reductions$ **do**
3: $P_{red} \leftarrow$ reduction of pointcloud P using an octree with voxel size *red*
4: **end for**
5: Write arff header to features.arff
6: **for all** points $p \in P_{reduce}$ **do**
7: $features \leftarrow$ empty list
8: $k_{Opts} \leftarrow$ empty list
9: **for all** $i \in \{0, 1, \ldots, k_{len} - 1\}$ **do**
10: $neighborhood \leftarrow$ nearest $k_{Maxs}[i]$ neighbors of point p in $P_{k_{Reductions}[i]}$
11: $k_{Opt} \leftarrow$ GetKOpt(p,$neighborhood$,$k_{Mins}[i]$,$k_{Deltas}[i]$)
12: $k_{Opts}.append(k_{Opt})$
13: $features.append($Calc3D2DFeatures(p,$neighborhood$,k_{Opt})$)$
14: **end for**
15: **for all** $i \in \{0, 1, \ldots, cyl_{len} - 1\}$ **do**
16: $neighborhood \leftarrow$ neighbors within a cylinder with radius $cyl_{Radii}[i]$ cm around point p in $P_{cyl_{Reductions}[i]}$
17: $features.append($CalcCylFeatures(p,$neighborhood$)$)$
18: **end for**
19: **if** semantic labels present **then** // *Training data*
20: $features.append(p.class)$
21: **end if**
22: Write *features* to features.arff
23: Write k_{Opts} to features_kOpt.txt
24: Write coordinates of p in uos format to features_points.3d
25: **end for**
26: **function** Calc3D2DFeatures(p,$neighborhood$,k) // *Calculate 17 3D/2D features*
27: **return** $(features_k^{3D}(p, neighborhood), features_k^{2D}(p, neighborhood))$
28: **end function**
29: **function** CalcCylFeatures(p,$neighborhood$) // *Calculate 4 cylindrical features*
30: **return** $features_\infty^{Cyl}(p, neighborhood)$
31: **end function**

Using three different reductions showed a reasonable tradeoff between classification accuracy and computation time. Choosing reduction scales of 2, 20, and 80 cm showed the best result with 72.42% accuracy. More reductions are imaginable but showed only small improvements while leading to higher computation times and a higher disk space usage for storing the features. In [45] a fixed number of ten neighbors is used on nine different scales for feature calculation.

Table 6.2: Impact of different and multiple reduction scales for feature calculation. The F_1 scores and accuracies of our random forest classifier are shown.

Reduction(s)	Terrain		Vegetation		Buildings	Hard Scape	Artefacts	Cars	Accuracy
	Man-m.	Nat.	High	Low					
0	0.946	0.356	0.758	0.335	0.578	0.011	**0.394**	0.132	**66.47%**
1	**0.947**	0.349	0.739	**0.337**	0.547	0.046	0.385	0.129	65.24%
2	0.939	**0.394**	0.744	**0.337**	0.571	0.047	0.363	0.141	65.68%
5	0.932	0.390	0.743	0.330	0.580	0.052	0.265	0.134	65.21%
10	0.916	0.359	0.749	0.307	0.583	**0.157**	0.227	0.123	64.26%
20	0.873	0.374	0.755	0.260	**0.708**	0.139	0.189	0.176	66.41%
40	0.848	0.315	**0.765**	0.187	**0.708**	0.135	0.166	0.164	65.20%
80	0.895	0.277	0.741	0.224	0.679	0.071	0.177	0.191	66.32%
160	0.767	0.048	0.688	0.258	0.618	0.028	0.220	**0.205**	58.31%
2, 10	0.923	0.431	0.764	**0.329**	0.603	0.064	0.350	0.135	65.84%
2, 20	0.933	**0.468**	0.777	0.318	0.627	0.070	0.324	0.130	67.34%
2, 80	**0.938**	0.371	**0.789**	0.325	**0.697**	**0.081**	**0.362**	**0.225**	**70.76%**
2, 10, 80	0.926	0.377	0.805	0.343	0.735	0.152	**0.382**	**0.212**	71.99%
2, 20, 80	**0.931**	**0.429**	**0.812**	0.321	0.740	0.122	0.379	0.196	**72.42%**
2, 8, 32	0.920	0.411	0.774	**0.385**	0.729	**0.158**	0.359	0.186	69.51%
5, 20, 80	0.926	0.402	0.804	0.340	**0.744**	0.125	0.317	0.202	72.09%
10, 20, 40	0.917	0.424	0.791	0.306	0.733	0.154	0.247	0.197	70.46%

The reductions were done with voxel sizes between 2.5 cm and 6.5 m. This leads to approximately the same spatial sizes as our neighborhoods reaching from ten neighbors in a 2 cm voxel reduced scan to 100 neighbors in a scan reduced with 80 cm voxel size.

Figure 6.3 depicts the size of points that are left after reducing a point cloud. Considering the logarithmic scale, 300 million points of Semantic3D's *sg27_station1* scan are reduced to approximately 43,000 points when using a voxel size of 80 cm.

Additionally, figure 6.4 illustrates the varying optimal neighborhood sizes in a whole laser scan for different reduction parameters. While in a detailed scan (2 cm voxel size) especially thin poles and sharp edges on cars show a high number of neighbors, on coarser reduction scales other areas prefer big neighborhoods. Interestingly mainly medium-sized objects like bushes and cars favor a high number of neighbors in a scan reduced with a voxel size of 80 cm. Even if it looks like heavily reduced clouds contain more points with a neighborhoods size close to 100, figure 6.5 shows that the distribution of different neighborhood sizes is very similar across all three reductions. The points with a big neighborhood size are just concentrated within a few small scan regions in these reductions.

For the previous results only one small cylindrical neighborhood of 25 cm on a scan reduced with voxel size 10 was used. Understandably, the previously found best point cloud reductions

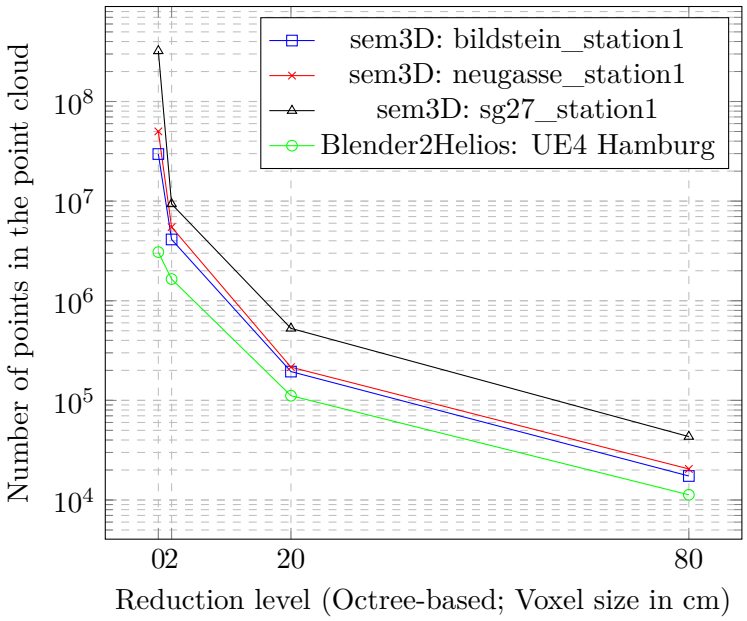


Figure 6.3: Size of octree-based reduced point clouds. Reductions were made with voxel sizes of 0, 2, 20, and 80 cm. Note the logarithmic scaling of the y axis.

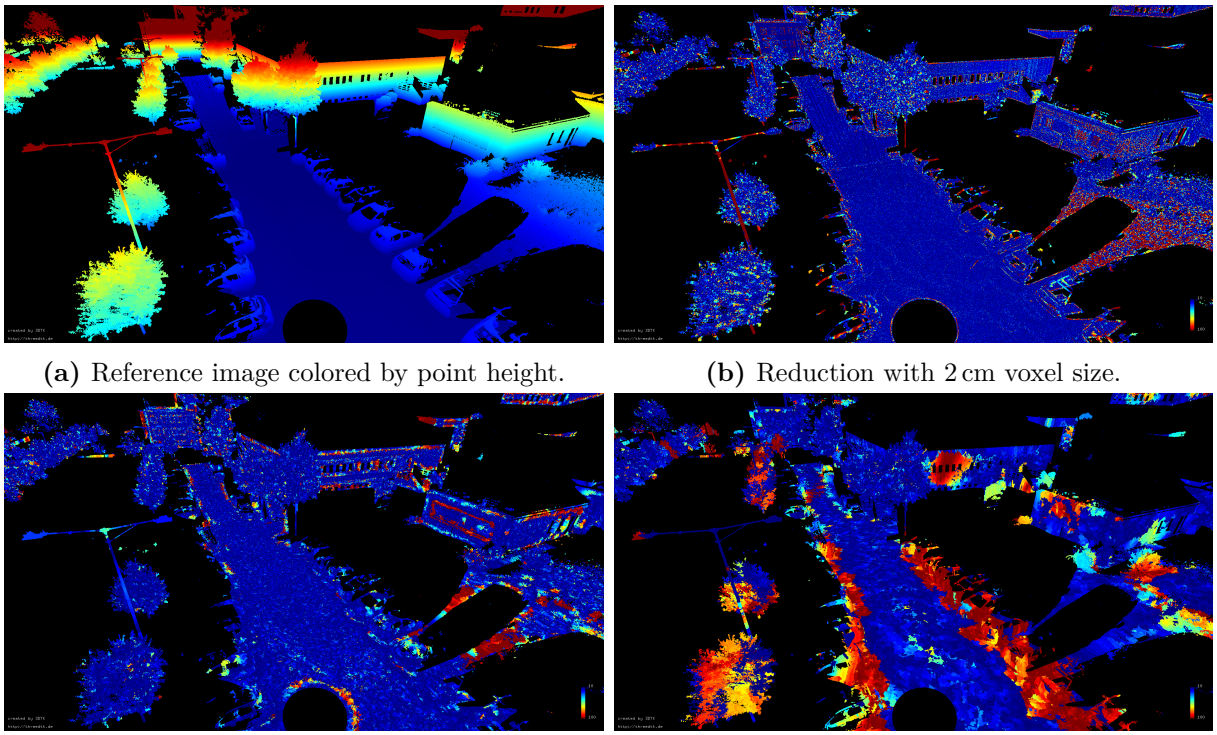
of 2, 20, and 80 cm for 2D/3D feature calculation are preferably also used for the cylindrical features. This minimizes the runtime as there is no need to calculate additional reductions. Table 6.3 shows the improvement of F_1 and accuracy scores when also using three approximated cylindrical neighborhoods. Due to time limitations the radii were chosen manually in a way that they contain approximately between 100 and 1,000 points in common outdoor scenes. This addition increased the overall accuracy by over 4% to 76.81%. Rechecking the possible impact of a higher number of trees in the forest showed again that the number of 100 trees is sufficient.

6.3 Parameter Tuning of the Classifiers

The following two sections describe the tuning of parameters for the random forest classifier and two SVM classifiers with different kernels.

Table 6.3: Impact of multiple cylindrical features and more trees in the forest.

# Trees	Cylindrical Features [Reduction(Radius)] in cm	Weighted Avg. F_1	Accuracy
100	10 (25)	0.752	72.42%
100	2 (25), 20 (150), 80 (500)	**0.786**	**76.81%**
500	2 (25), 20 (150), 80 (500)	0.784	76.61%



(a) Reference image colored by point height.

(b) Reduction with 2 cm voxel size.

(c) Reduction with 20 cm voxel size.

(d) Reduction with 80 cm voxel size.

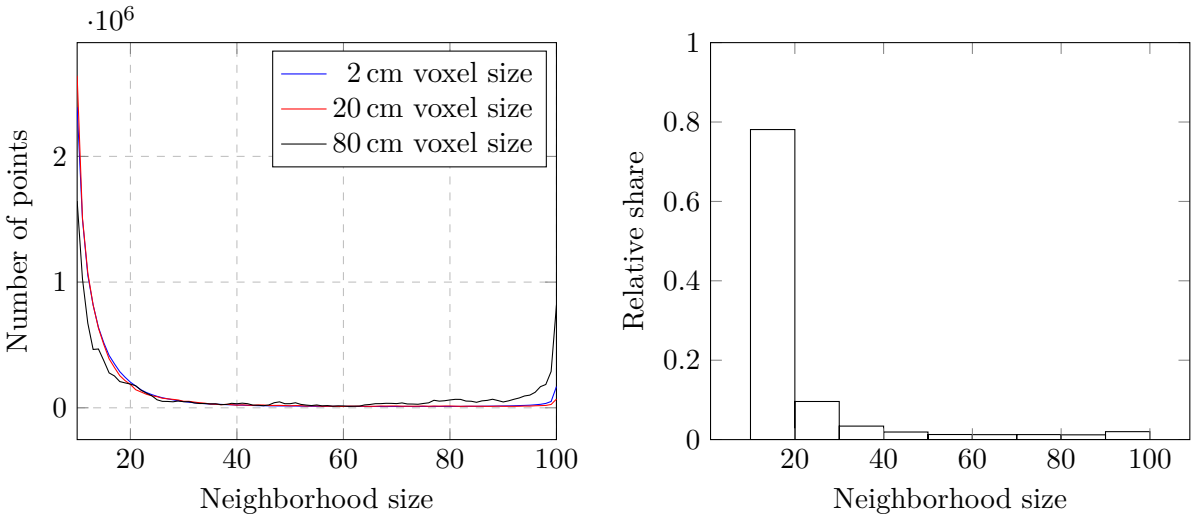
Figure 6.4: Illustration of optimal neighborhood sizes k_{Opt} on differently reduced scans. The color gradient indicates the found optimal neighborhood size from $k_{Min} = 10$ (blue) to red $k_{Max} = 100$ (red).

6.3.1 Random Forest

For the previously shown results a standard setup with 100 trees and a tree depth of four was used. After determining a good set of features, the classifier itself needed to be tuned. As indicated in table 6.3 before, we expected no further improvement when using more than 100 trees. The tables in appendix B support this estimation. Not only the different number of trees but also the impact of different random seeds for data subsampling and the creation of random forests is depicted there.

Therefore, the only parameter we additionally tuned was the tree depth. Figure 6.6 shows its influence. The overall accuracy increased from 75.96% (depth of 3) to 81.95% (depth of 63). Very small trees naturally discriminate some classes as they do not have enough branches to predict every class. The maximum of 63 was given by the number of 21 attributes we calculated on three different reductions.

To avoid overfitting we decided to keep the trees small and picked a tree depth of 20 for the later comparison. It corresponded to 81.59% overall accuracy on our validation data.



(a) Absolute distribution of k_{Opt} on different reduction scales on one of our real scans.

(b) Histogramm showing the relative distribution of k_{Opt} sizes in a scan reduced with voxel size 20 cm.

Figure 6.5: Distribution of optimal neighborhood sizes k_{Opt} on different reduction scales. Input was the outdoor scan created with the Riegl VZ-400 already shown in figure 6.4.

6.3.2 Support Vector Machines

In comparison to the results of the random forest, we also researched the accuracy of support vector machines on the calculated features. Two kernels (*RBF* and *Linear*) were evaluated with the LibSVM library in Weka. Due to the high run time complexity in the number of training samples, the training data was subsampled. The parameter selection had to be done with a training set of only 2,000 samples per class to allow checking a few hundred parameter settings. Readers that are new to SVMs may also consult [51] for a more detailed introduction to parameter tuning of SVMs.

Preprocessing for SVMs

In contrast to random forests, the magnitude between different features matters in a SVM. Thus, some kind of preprocessing had to be done. The standard approach for doing this is the *min-max normalization*. After application all feature values are within the range $[0, 1]$, where the minimum of all features values is mapped to 0 and their maximum is mapped to 1. The corresponding formula to calculate the normalized value $x'_{i,j}$ given the unnormalized value $x_{i,j}$ and the minimum $\min_j = \min_{i\in\{1,2,\ldots,n\}} x_{i,j}$ and maximum $\max_j = \max_{i\in\{1,2,\ldots,n\}} x_{i,j}$ of feature j is depicted in equation 6.8.

$$x'_{i,j} = \frac{x_{i,j} - \min_j}{\max_j - \min_j} \tag{6.8}$$

Due to outliers this normalization approach was not applicable for our features. To diminish the impact of outliers and as many of the features showed some kind of Gaussian dis-

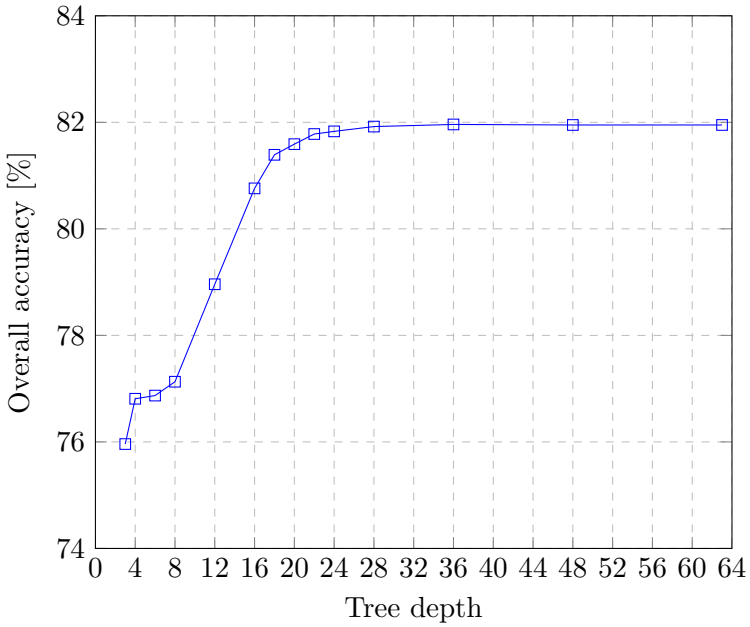


Figure 6.6: Influence of the tree depth on the random forest's accuracy.

tribution, we decided to perform *standardization*, also called *z-score normalization*. In this kind of preprocessing all values are scaled in such a manner that all features have zero-mean $\bar{x}'_j = \frac{1}{n}\sum_{i\in\{1,2,\ldots,n\}} x'_{i,j} = 0$ and unit-variance $\sigma'_j = 1$. The calculation is done as shown in equation 6.9 where $\bar{x}_j$ denotes the original mean of feature j and σ_j its standard deviation.

$$x'_{i,j} = \frac{x_{i,j} - \bar{x_j}}{\sigma_j} \tag{6.9}$$

With this kind of preprocessing most values stay within a small range around 0. Outliers are kept but scaled towards zero. One has to remember that the mean and standard deviation of the training set also has to be used when preprocessing data that the SVM is later applied on. Otherwise an error is introduced by the different distribution of the data.

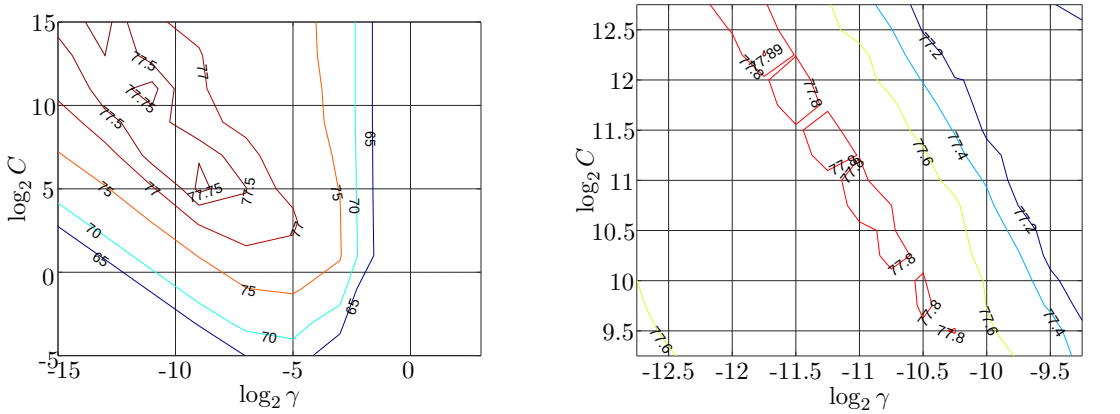
Radial Basis Function Kernel

SVMs with Radial Basis Function (RBF) kernel have two main parameters that have to be selected according to the data: C and γ. They are usually obtained by performing a grid search and using k-fold cross-validation.[51] Like in the case of random forests, we tested our parameters on a fixed data portion instead of cross-validation. This sped up the process and eliminated, as discussed before, the danger of over-fitting on the exactly seen object instances.

A grid search with 335 configurations for C and γ was performed. The results are shown in figure 6.7. In the coarse search, 110 configurations were tested. The exponents of C and γ to the basis of two were incremented by steps of two. In the finer search around the peak of $C = 2^{11}$ and $\gamma = 2^{-11}$, increments were reduced to 0.25 resulting in 225 more configurations

to check. The second peak at the coarse search was not taken into account as it lies closer to regions with lower accuracy. Choosing the more stable peak was reasonable.

As a result of the finer search, $C = 2^{12.25} = 4871$ and $\gamma = 2^{-12.5} \approx 0.00017263$ were chosen for all later experiments. This configuration showed the highest accuracy of over 77.89%.



(a) Coarse: $2^{-5} \leq C \leq 2^{15}$, $2^{-15} \leq \gamma \leq 2^{3}$ **(b)** Fine: $2^{9.25} \leq C \leq 2^{12.75}$, $2^{-12.75} \leq \gamma \leq 2^{-9.25}$

Figure 6.7: Grid search for parameters C and γ with LibSVM using a RBF kernel. The peak around $C = 2^{11}$ and $\gamma = 2^{-11}$ in the coarse search was broken down further in 0.25 increments in the exponent.

Linear Kernel

A classical linear C-SVM only needs the cost parameter C to be optimized. Typically, C is being searched within a range approximately between 2^{-5} and 2^{15}. As shown in figure 6.8, the accuracy was very stable, i.e. within a range of 0.25%, over a wide range of $2^{-1} \leq C \leq 2^{11}$. We therefore picked the median of these exponents and continued the experiments with $C = 2^5 = 32$.

6.4 Deep Learning: KPConv

6.4.1 Deep Learning for 3D Point Cloud Classification

There are many different approaches of using deep learning algorithms for classifying 3D point clouds. [105] splits these into four different categories. Some attempts rely on *projection methods* projecting the data to a regular grid structure. One can, for instance, render 2D images from a point cloud and apply common 2D classifiers afterwards [15, 64, 102, 103]. Other approaches suggest voxel-based methods using 3D grids [6, 73, 94].

Another category are *graph convolution networks*. Depending on the implementation, the convolution operator on a graph has been addressed in different ways. It can either be computed as a multiplication on its spectral representation [29, 120] or focus on the surface represented by the graph [20, 72, 76]. Compared to point convolutions, graph convolutions learn filters on edge relationships instead of the points' relative positions.

PointNet [87] is considered a milestone in *pointwise MLP networks*. A shared multilayer perceptron (MLP) is used on every point and individually followed by a global max-pooling.

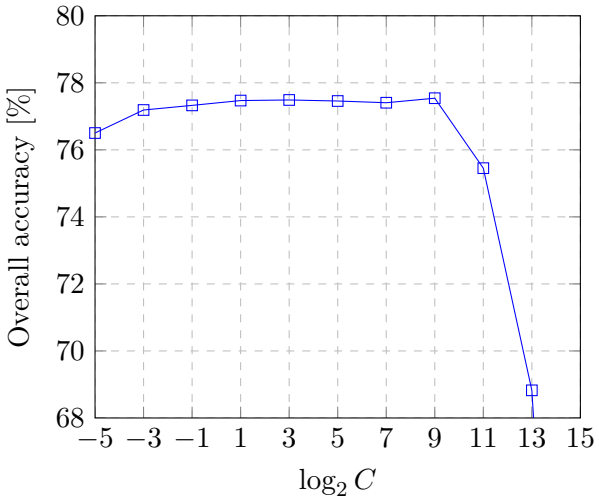


Figure 6.8: Influence of the cost parameter C on our SVM with linear kernel.

While the MLP learns a set of spatial encodings, the global part picks the maximal response among all the points for each of these encodings. Much other work, like [66, 88], was done to develop hierarchical architectures to aggregate local neighborhood information with MLPs.

KPConv belongs to the last category, the *point convolution networks*. Other well-known representatives are *Pointwise CNN* [118] and *SpiderCNN* [119]. In contrast to *KPConv*, *Pointwise CNN* locates the kernel weights with voxel bins. This means that it lacks flexibility. Furthermore, their approach involves a higher computational burden. In *SpiderCNN*, a kernel is defined as a family of polynomial functions with a different weight for each neighbor. Compared to *KPConv* this weight depends on the distance-wise order of neighbors instead of their actual distance. This makes their filters spatially inconsistent.[105]

6.4.2 Kernel Point Convolution (KPConv)

Kernel Point Convolution (KPConv) is a new design of point convolution described in [105]. It operates directly on point clouds without any intermediate representation. The used convolution concept is closely related to the convolution layers in CNNs already described in section 2.4.3. Due to the lack of a fixed grid structure, the convolution weights of KPConv are located in Euclidean space by so-called *kernel points*. These are applied on the input points close to them.

The authors describe their *KP-FCNN* as the applicable method for our point-wise classification task in 3D point clouds. It implements an encoder-decoder CNN consisting of convolutional blocks, designed like *bottleneck ResNet blocks* [46]. First, higher-order features are calculated step-by-step on reductions of the point cloud. Then, the original detailedness is achieved by nearest upsampling in the decoder part. Skip links [10, 92, 111] are used to transfer information between layers, skipping the network parts in between. The following descriptions heavily depend on [105] and its supplementary material [106].

The Kernel

In section 2.4.3 the typical definition of a convolution layer was illustrated. Instead of using a grid with weights, KPConv uses kernel points and a correlation function within spherical neighborhoods. Figure 6.9 illustrated this concept.

Further, let again $\boldsymbol{p_i}$ denote 3D points of a point cloud P. Corresponding features from $\boldsymbol{F} \in \mathbb{R}^{n \times d}$ are denoted as $\boldsymbol{f_i}$. The spherical neighborhood with radius r around a point $\boldsymbol{p} \in \mathbb{R}^3$ is defined by $N_p = \{\boldsymbol{p_i} \in P |\ ||\boldsymbol{p_i} - \boldsymbol{p}|| \leq r\}$. Equation 6.10 denotes the general point convolution of $\boldsymbol{F}$ using a kernel $\boldsymbol{g}$ at point $\boldsymbol{p}$.

$$(\boldsymbol{F} * \boldsymbol{g})(\boldsymbol{p}) = \sum_{\boldsymbol{p_i} \in N_p} g(\boldsymbol{p_i} - \boldsymbol{p}) \boldsymbol{f_i} \tag{6.10}$$

Starting from here, $\boldsymbol{y_i}$ denotes the vector $\boldsymbol{p_i} - \boldsymbol{p}$ for clarity. Similar to 2D convolutions, g is expected to apply different weights within an local area. For this, a neighborhood ball with radius r is defined by $B_r = \{\boldsymbol{y} \in \mathbb{R}^3 |\ ||\boldsymbol{y}|| \leq r\}$.

Given the k kernel points $\{\tilde{\boldsymbol{p}}_{\boldsymbol{j}} |\ j < k\} \subset B_r$ and the corresponding weight matrices $\{\boldsymbol{W_j} |\ j < k\} \subset \mathbb{R}^{d_{in} \times d_{out}}$ to map the features from a layer's input dimension d_{in} to a layer's output dimension d_{out}, the kernel function g is given by equation 6.11.

$$g(\boldsymbol{y_i}) = \sum_{j<k} h(\boldsymbol{y_i}, \tilde{\boldsymbol{p}}_{\boldsymbol{j}}) \boldsymbol{W_j} \tag{6.11}$$

The correlation h between two points should be high if both points are close together. Inspired by the bilinear interpolation described in [27], the authors of KPConv decided to use the linear correlation shown in equation 6.12. σ regulates the impact of the distance between the two points and is chosen according to the input density.

$$h(\boldsymbol{y_i}, \tilde{\boldsymbol{p}}_{\boldsymbol{j}}) = \max(0,\ 1 - \frac{||\boldsymbol{y_i} - \tilde{\boldsymbol{p}}_{\boldsymbol{j}}||}{\sigma}) \tag{6.12}$$

Another critical aspect is the alignment of the kernel points. The authors of KPConv propose two approaches. The first uses *rigid* kernel point positions. A number of k kernel points is spread across the sphere by applying repulsive forces and solving the optimization problem. One point is constrained to stay in the middle. This rigid version is extremely efficient.

On the other hand, there are *deformable* kernels. As the kernel function g is differentiable with respect to $\tilde{\boldsymbol{p}}_{\boldsymbol{j}}$, they are learnable parameters. The network learns a set of k different shifts $\boldsymbol{\Delta_p}$ for every convolution position $\boldsymbol{p} \in \mathbb{R}^3$. This results in the new convolution function shown in equation 6.13 with the kernel function g_{deform} as depicted in equation 6.14. The offsets $\boldsymbol{\Delta_p^k}$ are defined as the output of a rigid KPConv mapping d_{in} input features to $3k$ values. [105] shows that the descriptive power of deformable kernels is higher than that of rigid kernels – especially for a small number of kernel points.

$$(\boldsymbol{F} * \boldsymbol{g})(\boldsymbol{p}) = \sum_{\boldsymbol{p_i} \in N_p} g_{deform}(\boldsymbol{y_i}, \boldsymbol{\Delta_p}) \boldsymbol{f_i} \tag{6.13}$$

$$g_{deform}(\boldsymbol{y_i}, \boldsymbol{\Delta_p}) = \sum_{j<k} h(\boldsymbol{y_i}, \tilde{\boldsymbol{p}}_{\boldsymbol{j}} + \boldsymbol{\Delta_p^k}) \boldsymbol{W_j} \tag{6.14}$$

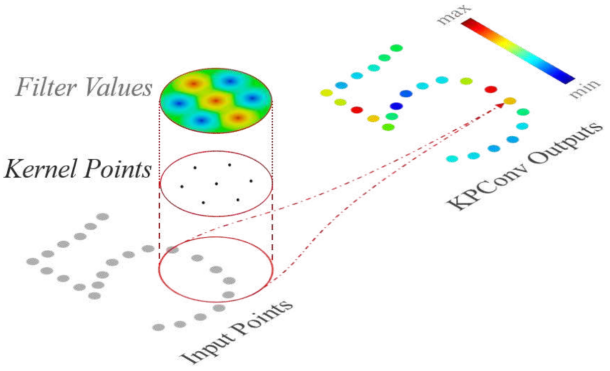


Figure 6.9: Illustration of KPConv. A kernel point convolution is applied on scalar input points. Each kernel point has a filter weight. The output at the current location is the sum of the weighted features. Taken from [105].

Our Setup

In [105] KPConv was already applied on numerous datasets. One of them is the Semantic3D dataset our RF classifiers was tuned with. The source code is available at `https://github.com/HuguesTHOMAS/KPConv`. In the available file for training (*training_Semantic3D.py*) their optimized parameters can be reviewed. It describes the 18-layer encoder-decoder network and the exact kernel parameters. 15 kernel points are used and a subsampling with a voxel size of 6 cm is performed at the beginning. The network's learning rate decreases slowly, starting with a value of 10^{-2}. Cross-entropy loss is calculated after a batch with a size of ten while the training ends after 500 epochs.

As the parameters were already optimized by the authors, we decided to copy these settings. We will skip further details here as [105, 106] provide all the information about the architecture. Readers are also invited to examine the well-arranged published source code files for details.

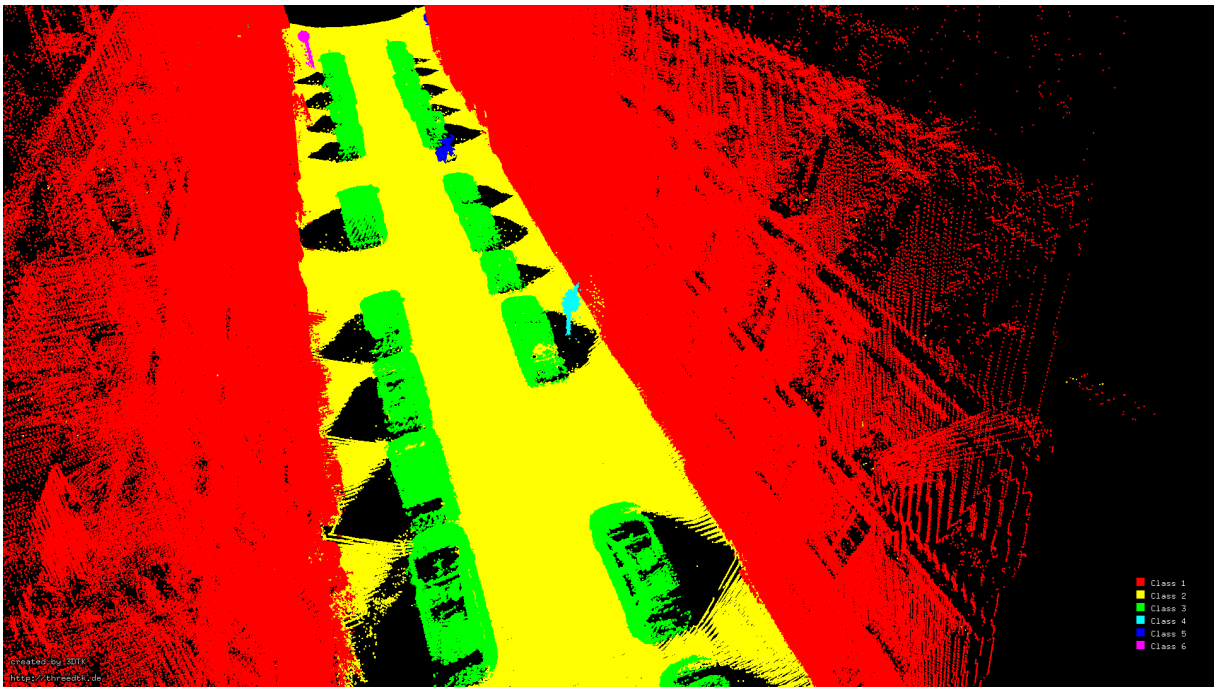
For the following comparison we only adapted the handling of additional features in the data. Under fair conditions KPConv was only trained on three-dimensional coordinates without additional color or reflectance information.

Chapter 7

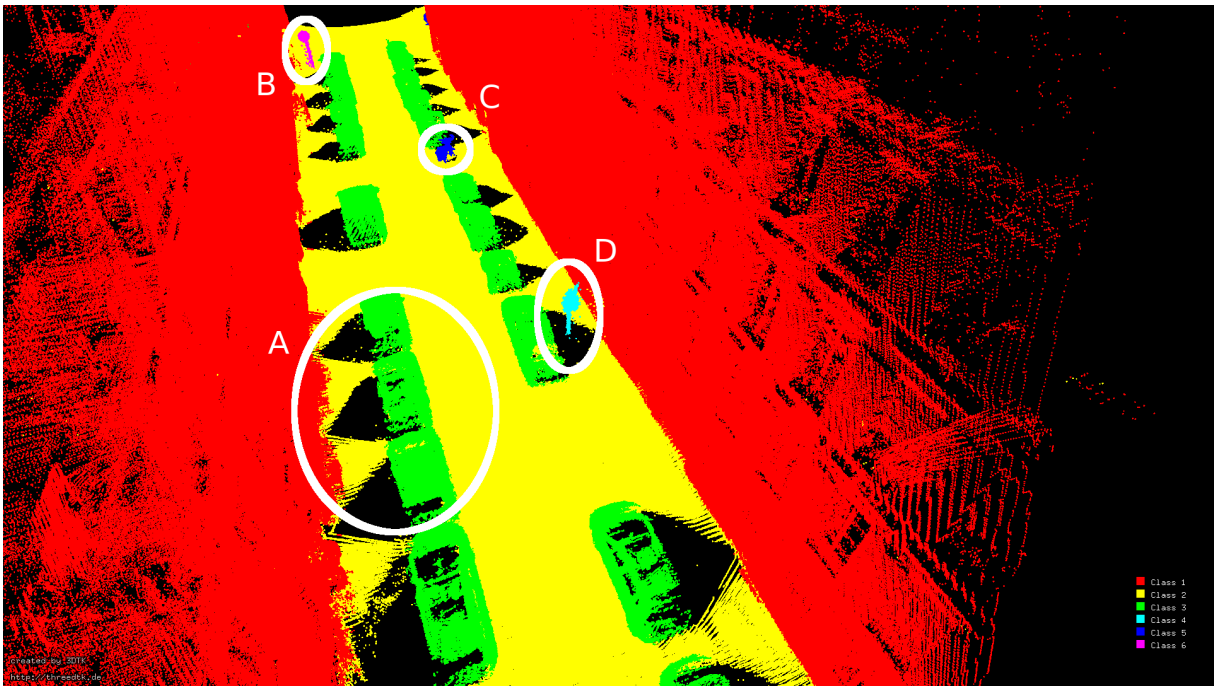
Results and Comparison

This chapter shows a comparison of the different classifiers applied on the four datasets. Section 7.4 presents the results we achieved when participating in the Semantic3D challenge. A coarse runtime comparison between the two methods with the highest accuracy, our RF approach and KPConv, is additionally given in section 7.5. The settings of the feature calculation procedure were chosen as previously shown: $k_{Red} = cyl_{Red} = (2, 20, 80)$ and $cyl_{Radii} = (25, 150, 500)$. Random forests with 100 trees and a tree depth of 20 were trained. The scores of the original RF implementation were taken from the original paper [115] for reference.

The figures 7.1 to 7.4 show the real class labels of the *Paris-rue-Madame* and the *Sim2Real* datasets including the results of our random forest and the KPConv classifiers. The results are discussed in the following sections belonging to the datasets.

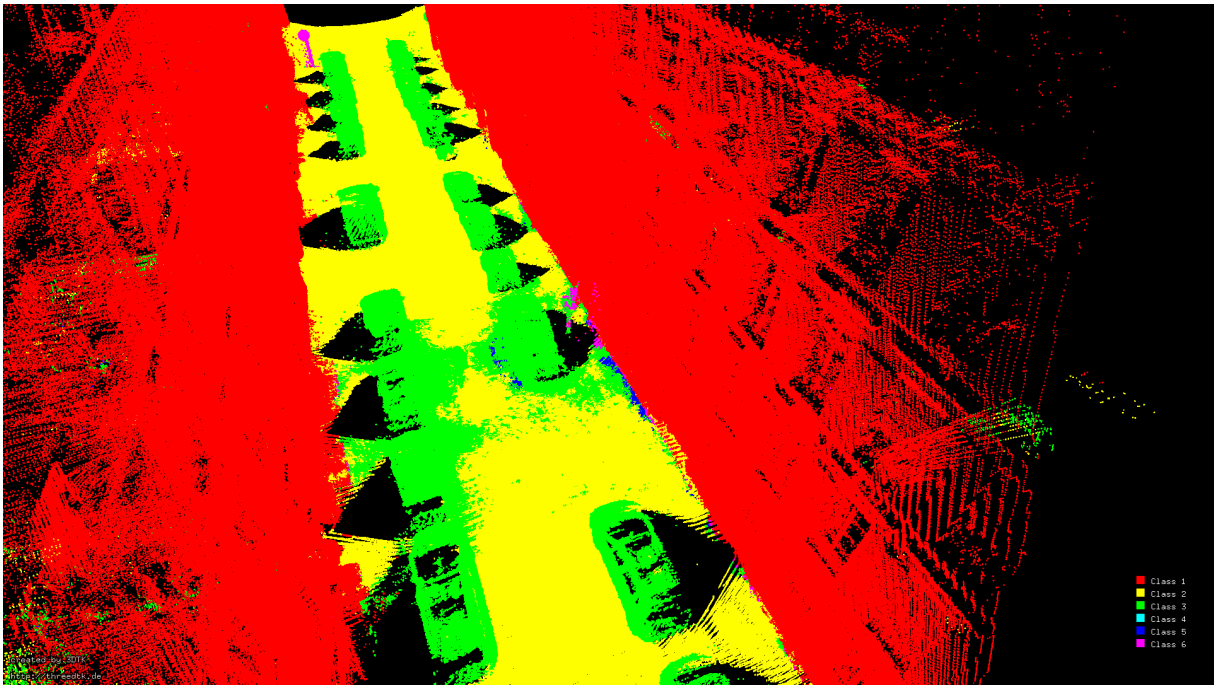


(a) Semantic labels in the Paris-rue-Madame test data.

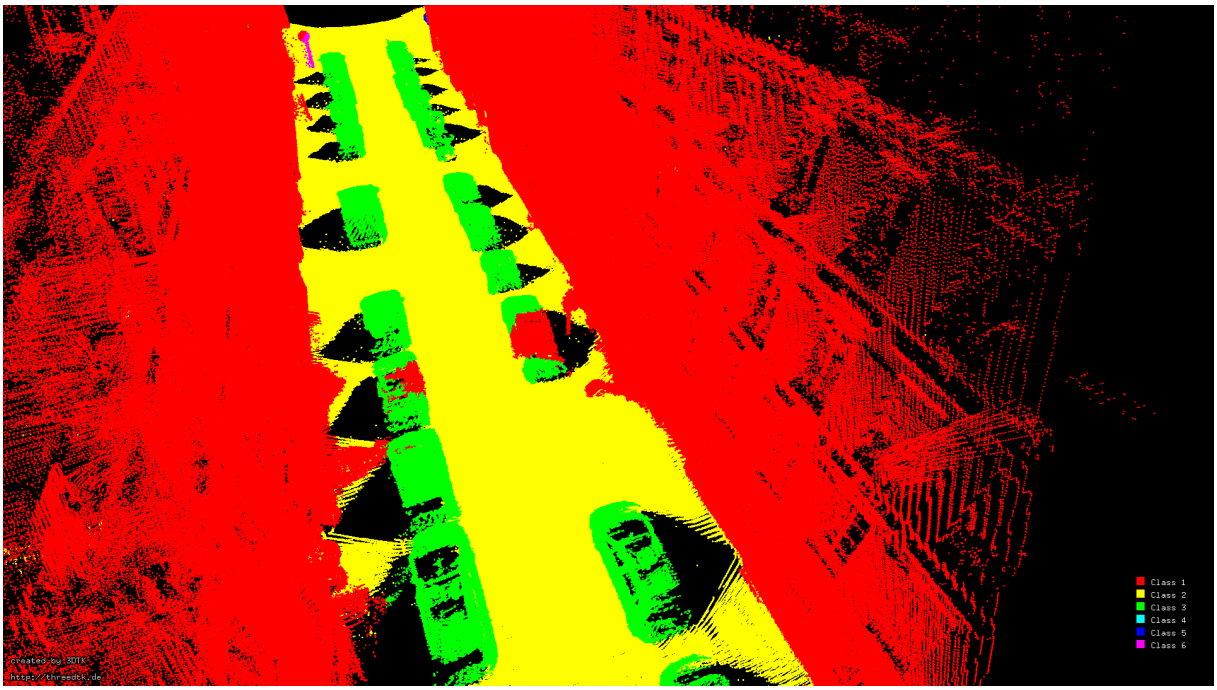


(b) White circles indicate regions that are later discussed in the text.

Figure 7.1: Paris-rue-Madame's labeled test data for reference. Colors indicate class affiliations. *Red*: facade, *yellow*: ground, *green*: cars, *light blue*: pedestrians, *dark blue*: motorcycles, *rose*: traffic signs.

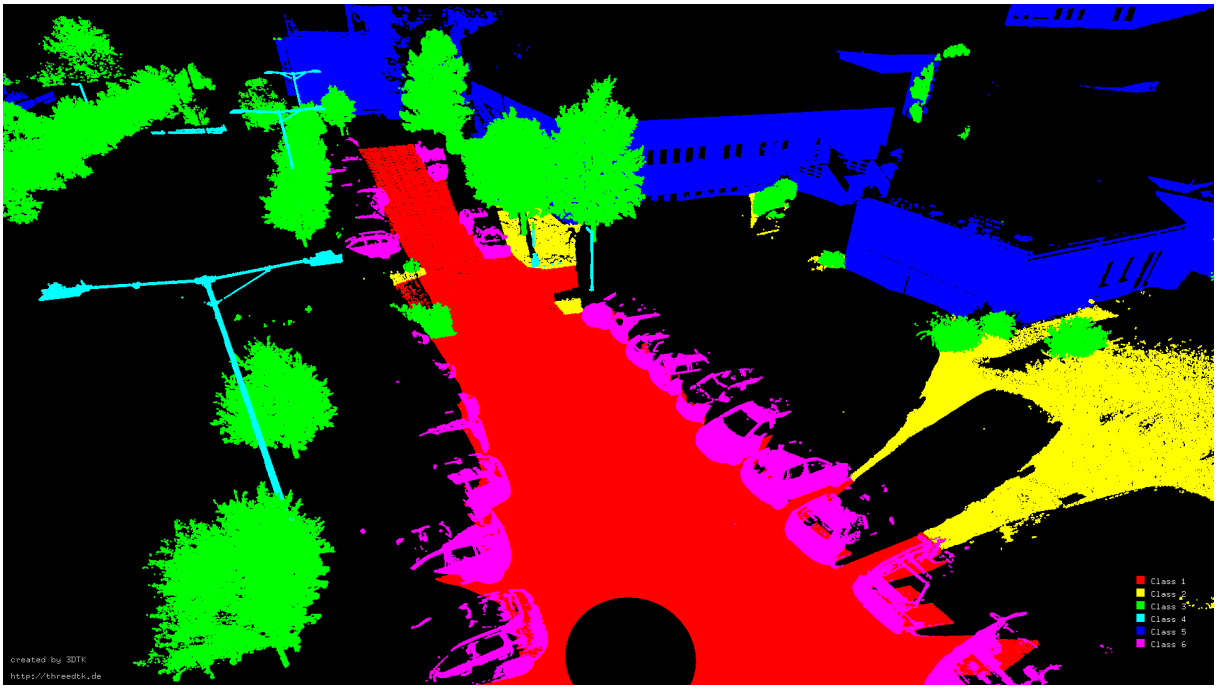


(a) Output of our random forest classifier.

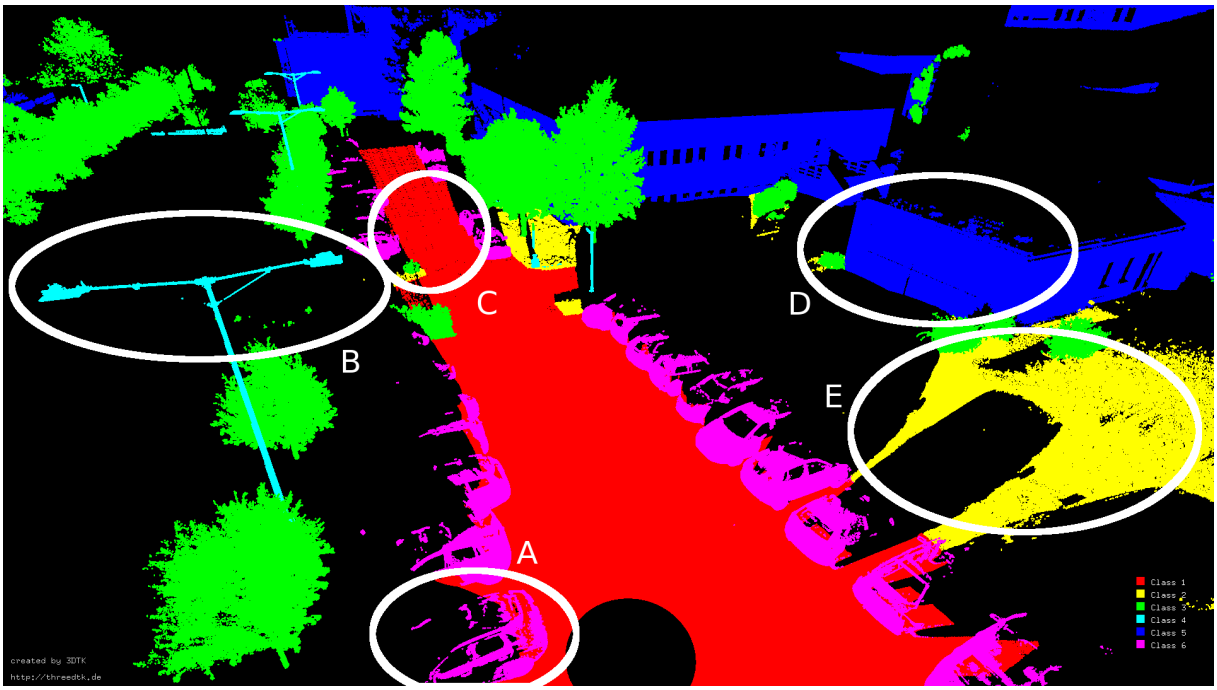


(b) Output of KPConv.

Figure 7.2: Output of our RF and KPConv applied on the Paris-rue-Madame dataset. Colors indicate class affiliations. *Red*: facade, *yellow*: ground, *green*: cars, *light blue*: pedestrians, *dark blue*: motorcycles, *rose*: traffic signs.

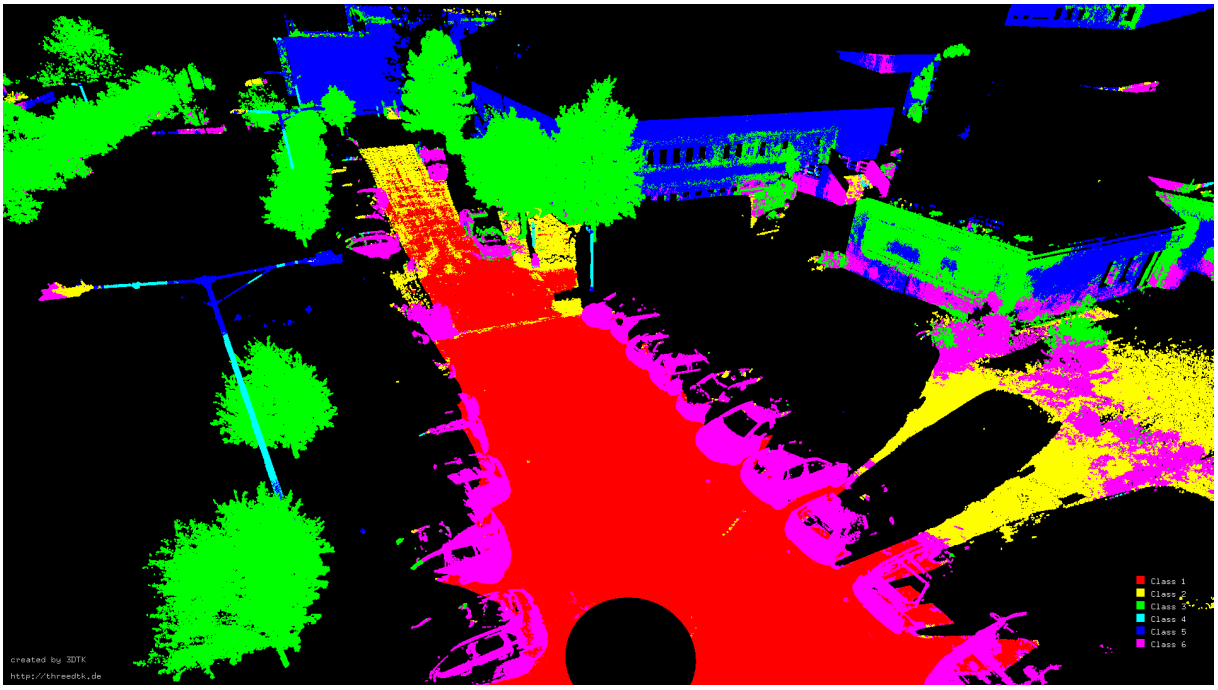


(a) Semantic labels in the Sim2Real test data. Annotation was done manualy.

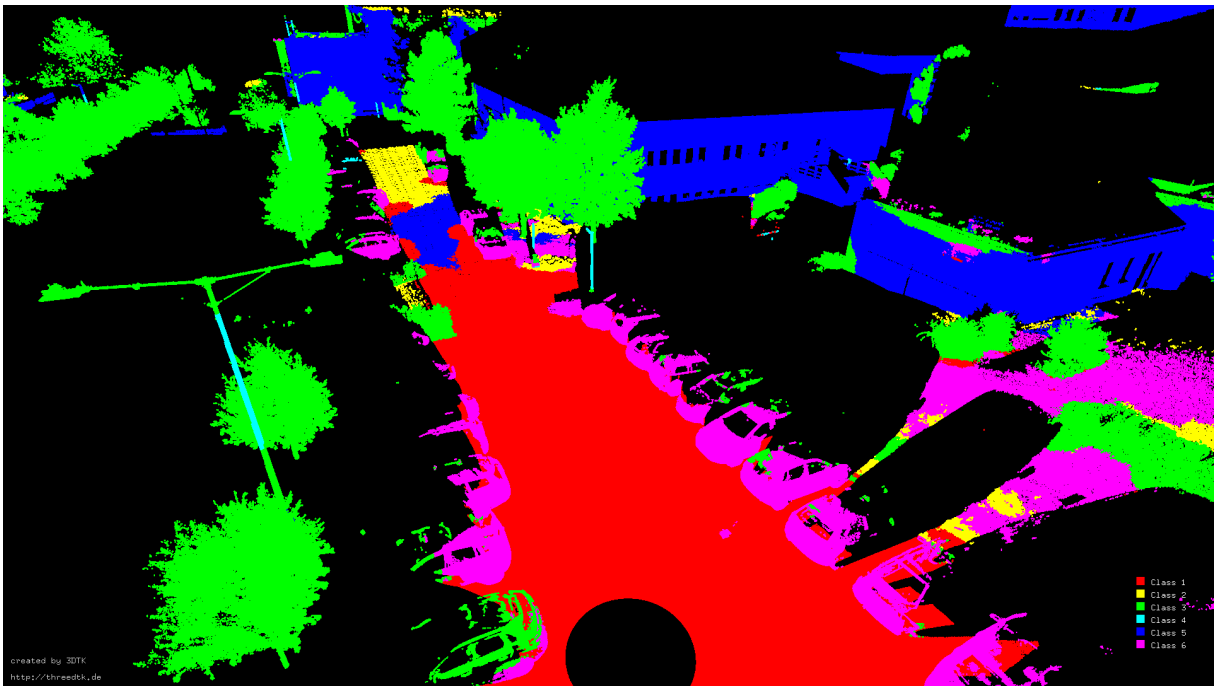


(b) White circles indicate regions that are later discussed in the text.

Figure 7.3: Sim2Real's labeled test data for reference. Colors indicate semantic class affiliations. *Red*: man-made ground, *yellow*: natural ground, *green*: vegetation, *light blue*: object, *dark blue*: facade, *rose*: car.



(a) Output of our random forest classifier.



(b) Output of KPConv.

Figure 7.4: Output of our RF and KPConv applied on the Sim2Real dataset. Colors indicate semantic class affiliations. *Red*: man-made ground, *yellow*: natural ground, *green*: vegetation, *light blue*: object, *dark blue*: facade, *rose*: car.

7.1 Oakland

Applied on the Oakland dataset our RF classifier showed an accuracy of 95.27% which is around 3% higher than the results using the original single-scale approach described in [115]. Our approach also minimizes the assumptions made, like a consistent height of the laser scanner. The detailed result of our random forest is shown in table 7.1. Class-wise F_1 scores and the overall accuracies of the different classifiers are printed in table 7.2. While both SVMs' performances were far behind, KPConv performed slightly better than our RF approach. Recall and precision values as well as detailed confusion matrices can be found in appendix D.1.

It has to be noted that this dataset only contained 1,086 points belonging to class *pole*. The RF and SVM classifiers were therefore only trained on 5,430 equally distributed points.

Table 7.1: Performance of our RF on the Oakland dataset. Overall accuracy: 95.27%.

Our Label	Orig. Label	Class Name	Precision	Recall	F-Measure
1	1004	Scatter/Misc	0.953	0.922	0.937
2	1100	Wire	0.101	0.894	0.181
3	1103	Pole	0.376	0.751	0.501
4	1200	Load/Bearing	0.999	0.994	0.996
5	1400	Facade	0.893	0.696	0.782
Weighted Avg.			0.974	0.953	0.961

Table 7.2: Comparison of F_1 scores and accuracies of the classifiers on the Oakland dataset.

	Class F_1 Score					Overall
Classifier	**1**	**2**	**3**	**4**	**5**	**Accuracy**
Original RF	0.878	0.165	0.364	0.978	0.749	92.2%
Our RF	0.937	0.181	**0.501**	**0.996**	0.782	95.3%
Our SVM_{RBF}	0.772	0.039	0.098	0.894	0.420	75.8%
Our SVM_{Linear}	0.811	0.030	0.107	0.893	0.336	75.9%
KPConv	**0.966**	**0.186**	0.429	0.992	**0.813**	**95.7%**

7.2 Paris-rue-Madame

We used *scan 1_2* for training and *scan 1_3* for testing. Due to the small number of pedestrians in the training data, the data was subsampled to 3,656 examples per class for our RF and SVM classifiers. Compared to the original paper we achieved an over 6% higher accuracy of 96.44% as presented in table 7.3. Another 2% were achieved by KPConv.

Table 7.3: Comparison of F_1 scores and accuracies of the classifiers on the Paris-rue-Madame dataset.

	Class F_1 Score						**Overall**
Classifier	**1**	**2**	**3**	**4**	**5**	**6**	**Accuracy**
Original RF	0.960	0.932	0.672	**0.036**	0.206	0.105	90.1%
Our RF	0.983	0.965	0.897	0.000	0.202	0.304	96.4%
Our SVM_{RBF}	0.973	0.924	0.859	0.000	0.119	0.033	91.7%
Our SVM_{Linear}	0.977	0.914	0.864	0.000	0.049	0.069	91.5%
KPConv	**0.989**	**0.988**	**0.972**	0.000	**0.733**	**0.767**	**98.6%**

Figure 7.2 shows the outputs of our random forest and KPConv after applying them to the test data shown in figure 7.1. Region A in the figure shows an area where the high recall value belonging to the class *cars* of our random forest classifier can be seen. However, also the points belonging to the ground around the cars is assigned the same label. In constrast, KPConv labels cars more conservative leading to a higher precision score but introducing some errors where *facades* are predicted. The traffic sign in region B is detected with a high accuracy by both methods. KPConv shows only a small mistake where is detects *facade* points. The motorcycle in area C was not detected in both outputs. KPConv, however, detected other motorcycles that are not presented in the illustration with a higher accuracy. The pedestrian in sector D was ignored by both approaches.

Especially the bad matching of pedestrians has to be noted. Precision and recall values are zero for this class as shown in table 7.4. We interpret this as an over-fitting on the seen object instances due to the fact that only three pedestrians were part of the training data. Especially by the used optimal neighborhood definition, all points of a pedestrian are described by roughly the same features. Thus, oversimplified, the data can be seen as only three data points with a higher weight. For instance in class 1, 181 different facades were also down-sampled to 3,656 points. The variance of these points is much higher resulting in a classifier that recognizes many different kinds of facades. Lowering the tree depth by a factor of 1/2 to allow a higher degree of generalization did, however, not show any improvement.

Confusion matrices and more precision/recall values can be found in appendix D.2.

Table 7.4: Performance of our RF on the Paris-rue-Madame dataset. Overall accuracy: 96.44%.

Our Label	Orig. Label	Class Name	Precision	Recall	F-Measure
1	1	Facade	0.984	0.982	0.983
2	2	Ground	0.994	0.937	0.965
3	4	Cars	0.816	0.995	0.897
4	9	Pedestrians	0.000	0.000	0.000
5	10	Motorcycles	0.291	0.154	0.202
6	14	Traffic signs	0.200	0.635	0.304
Weighted Avg.			0.968	0.964	0.965

7.3 Sim2Real

The simulated laser scan created with Blender, Blender2Helios, and Helios was subsampled to 16,900 examples per class for training our RF/SVMs. Table 7.5 shows the detailed performance of our RF classifier. Evaluation was done with the manually labeled real scan from the parking lot in Würzburg.

Table 7.5: Performance of our RF classifier on the Sim2Real dataset. Overall accuracy: 94.21%.

Label	Class Name	Precision	Recall	F-Measure
1	Manmade Ground	1.000	0.973	0.986
2	Natural Ground	0.784	0.762	0.773
3	Vegetation	0.876	0.961	0.916
4	Lamps/Railings	0.893	0.304	0.454
5	Buildings/Walls	0.959	0.650	0.775
6	Cars	0.827	0.986	0.900
Weighted Avg.		0.947	0.942	0.940

Results of the different classifiers are shown in table 7.6. Our RF and KPConv reached the same overall accuracy of 94.21%. Only the SVM with RBF kernel showed a very similar accuracy. The linear kernel was approximately 8% behind KPConv and our approach. It has to be noted that this table misses a comparison with the original RF implementation as this dataset was created on our own and no reference values exist. Confusion matrices and detailed scores are again attached in the appendix (D.3).

Figure 7.3 shows an excerpt of the test data. It also highlights regions that were labeled very differently by our random forest approach and KPConv. Their outputs are printed in figure 7.4. Region *A* shows a car that was correctly classified by us, but was treated like vegetation by KPConv. The recognition of area *B*, the upper part of a street light, was incorrectly classified by both methods. Sector *C* shows paved ground. It is likely that a higher variance in the handcrafted features, potentially originated by the higher distance to the scanner, led to the

Table 7.6: Comparison of F_1 scores and accuracies of the classifiers on Sim2Real.

	Class $\mathbf{F_1}$ Score						Overall
Classifier	**1**	**2**	**3**	**4**	**5**	**6**	**Accuracy**
Our RF	0.986	**0.773**	**0.916**	**0.454**	0.775	**0.900**	**94.2%**
Our SVM_{RBF}	0.989	0.662	0.885	0.239	0.786	0.890	92.5%
Our SVM_{Linear}	0.961	0.366	0.892	0.121	0.835	0.727	86.2%
KPConv	**0.990**	0.364	0.888	0.373	**0.946**	0.876	**94.2%**

classification as natural ground. We have no explanation why KPConv detected a facade here. However, in part *D*, KPConv detects the building with a much higher accuracy. Area *E* is probably the most difficult section. The grassland has an irregular incline and is peppered with smaller plants. Big parts of it were correctly classified by the random forest. KPConv, in constrast, heavily labeled the points as cars or vegetation.

7.4 Semantic3D Challenge

Semantic3D is the dataset containing the most points for training and testing. As both SVM approaches were already clearly outperformed on the other datasets, they were ignored here for time reasons.

Our RF To tune the parameters, an additional split of the supplied training data was used. Now, all the prelabeled data was used to create an equally distributed set of features to train a classifier with best performance.

KPConv As previously mentioned, KPConv is already provided with a training procedure for the Semantic3D challenges. As the authors of [105] already put much effort into tuning, this was just adapted to our needs. The only change that was made was the removal of the color information. Our results show that this change did not impact the classification accuracy significantly. In this case the classifier even got slightly better.

The results of our submissions to Semantic3D's reduced-8 challenge are given in table 7.7. For comparison the original KPConv results that used color information are also depicted. F_1 scores and accuracies are shown in table 7.8. Confusion matrices and precision and recall values can be found in appendix D.4.

Table 7.7: Results of the Semantic3D reduced-8 submissions: Intersection over Union.

	Colors	**Intersection over Union for Class**								**Average**
Classifier	**Used?**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**IoU**
Our RF	No	0.886	0.839	0.595	0.371	0.861	0.187	0.203	0.398	0.542
KPConv	No	**0.982**	**0.905**	0.814	0.373	0.943	0.319	0.609	**0.824**	0.721
KPConv	Yes	0.909	0.822	**0.842**	**0.479**	**0.949**	**0.400**	**0.773**	0.797	**0.746**

Table 7.8: Results of the Semantic3D reduced-8 submissions: F_1 scores and accuracies.

	Colors	**Class F_1 Score**								**Overall**
Classifier	**Used?**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Accuracy**
Our RF	No	0.940	0.912	0.746	0.541	0.926	0.315	0.337	0.569	86.49%
KPConv	No	**0.991**	**0.950**	0.897	0.543	0.971	0.484	0.757	**0.903**	**93.51%**
KPConv	Yes	0.952	0.902	**0.914**	**0.648**	**0.974**	**0.571**	**0.872**	0.887	92.86%

7.5 Run Times

The calculations were usually done on different hardware. While a server with many CPU cores was used to improve performance of the feature calculation in our RF approach, another machine was used to train the random forests. Yet another system with a modern graphics card allowed running the KPConv implementation.

To be able to do a comparison, the training and testing procedures of our classical random forest approach and KPConv on the Sim2Real dataset were repeated on similar hardware. Due to the corona pandemic at the time this thesis was written, only a laptop with external GPU was available.

The used *Lenovo Yoga 720-15IKB* was equipped with an *Intel Core i5-7300HQ* CPU (4x 2.50 GHz) and 24 GB of RAM. An *Asus GeForce RTX 2080 Ti ROG STRIX* with 11 GB of video memory was plugged into a *Razer Core X* external GPU graphics card enclosure. The Thunderbolt connection was caught to be a small bottleneck and only allowed utilizing the GPU up to approximately 70-80%. Of course, modern workstations will perform better than our hardware. However, the times shown in table 7.9 already provide a coarse reference.

Table 7.9: Run times of our RF and the KPConv classifier.

			Time Taken (Approx.)	
Task	**Number of Points**	**Step**	**Our RF**	**KPConv**
Training	2 million	Feature calculation	40 seconds (100,000)[1,2]	7 hours
		Classifier training	40 seconds (100,000)[3]	
Training	10 million	Feature calculation	3 minutes (400,000)[1,4]	13.5 hours
		Classifier training	4 minutes (400,000)[3]	
Testing	10 million	Feature calculation	1 hour	1 hour
		Applying classifier	9 minutes	

[1] These values were roughly estimated as our current implementation only supports the feature calculation on whole 3D scans. The number in brackets shows the number of points that we were able to choose equally distributed from the whole scan for training.
[2] Feature calculation on all 2 million points took 11 minutes.
[3] The random forests were only trained on the equally distributed number of points.
[4] Feature calculation on all 10 million points took roughly one hour.

The numbers indicate that calculating handcrafted features and training a random forest is much faster than the neural network approach. It has to be mentioned that KPConv was also optimized for the huge Semantic3D dataset. While we trained the network for 500 epochs, the loss was very stable after 200 to 300 epochs, depending on the dataset. Training times of our RF solution adapt better to varying dataset sizes. By changing the learning rate and adapting the stop criterion a speedup of the KPConv learning process seems likely.

Chapter 8

Conclusions and Future Work

The main objective of this research was the comparison of different approaches for semantic classification in 3D point clouds. One can see that classical methods with handcrafted features still have a right to exist. With our optimization of using multiple reduced scans to approximate different neighborhood sizes they showed very similar results compared to the deep learning approach on small- to medium-sized datasets. Their short runtime for training makes them an excellent choice for creating a good baseline or for choosing a reasonable set of training data. Due to their low hardware requirements, they can be used on cheap and even mobile hardware. The comparison of different classifiers showed that random forests perform better than SVMs with linear or RBF kernels.

On the big *Semantic3D* dataset the deep learning approach performed better. We justify this by its capability of learning a more complex model. This comes with the downside of long training times and increased hardware prerequisites. The used *KPConv* network needed approximately nine gigabytes of graphics memory. Expensive and power-consuming hardware is needed to apply this method.

All the written code is released publicly to allow the reproduction of our results. While the implementations allow users to change many parameters directly, one can easily adapt the algorithms to his or her needs.

With the development of *Blender2Helios*, an – in our opinion necessary – interface between an easily usable 3D suite and the LiDAR simulation software *Helios* was created. Due to the good compatibility of *Blender*, many existing scenes can now be converted to 3D point clouds. As labeling in *Blender* is much easier than in the point cloud, we hope to promote the creation of highly detailed and labeled training data for the learning of classifiers. In this thesis we showed with the *Sim2Real* dataset that artificial data can be used to train classifiers that are later used in real-world applications.

Future Work Many different neighborhood definitions and sets of handcrafted features exist. It is conceivable that other combinations achieve higher accuracies than the ones we have tested. [104], for instance, recommends using spherical neighborhoods with a specific radius instead of using the k nearest neighbors. The authors justify this by a lower sensitivity to varying densities. As we perform reductions on the point clouds this impact is expected to be relatively

small. Additionally, in almost every publication a slightly different set of features is used. One possible enhancement is the extension of *scan2features* to allow combining a variation of them. Furthermore, as shown in chapter 7, an improvement when calculating features for classifier training can be implemented. It is reasonable to calculate features for an equally distributed set of points only. This significantly reduces the time needed for training. On the other hand, all training instances could be weighted. It is likely that the classifiers generalize better after seeing a higher number of different samples.

Since the classical and the deep learning approach showed different errors, one may combine the methods and do a vote for the correct class. We are highly interested in the results of this setup.

In addition, we want to promote the use of *Blender2Helios* again. For instance in the field of computer games, large and highly-detailed 3D scenes were already created. In combination with our published preprocessing script, these can easily be converted to labeled 3D scans to create training data. One can also use our tool to create laser scans of own scenes. This is especially helpful when training data of similar scenes is not or only sparsely available. We hope to see some applications of our tool in future publications.

Appendices

Appendix A

Blender2Helios Code Schema

Algorithm 6 Blender2Helios - Schema of exporting a Scene (Part 3)

1: *// The following functions are only briefly described for readability reasons*
2: **function** XMLSCENEHEAD( ) ... *// Returns a default XML header for a scene*
3: **end function**
4:
5: **function** XMLSCENEFOOT( ) ... *// Returns a default XML footer for a scene*
6: **end function**
7:
8: **function** XMLSURVEY( ) ... *// Returns XML code for a default 360° scan survey*
9: **end function**
10:
11: **function** TOTEXT(elements) ... *// Returns string of values*
12: *"*$e_1 - e_2 - \cdots - e_n$*"* ($e_i \in elements$)
13: **end function**
14:
15: **function** SCALE2ORIGIONAL(object) ... *// Returns dimensions of object as if scaling of first dimension was 1 (object.dimensions/object.scale[0]). For instance returns [100,200,300] for object with dimensions [200,400,600] when scaled by [2,y,z] for any y,z*
16: **end function**
17:
18: **function** OBJECT2XML($cName$, $fileName$, $translation$, $rotation$, $scale$) ... *// Returns Helios compatible XML string that represents one object.*
19: **end function**
20:
21: **function** QUATERNION2RPY($quaternion$) ... *// Returns roll-pitch-yaw of quaternion*
22: **end function**

Appendix B

Impact of Different RF Configurations and Random Seeds

Table B.1: Different configurations for randomness and tree size and the resulting classification accuracy. Trained and tested on our own split of the Semantic3D dataset.

Conf. #	**Forest Size**	**Seed (Subsampl.)**	**Seed (RF)**	**Accuracy**
1	10	1	1	64.90%
2	25	1	1	65.49%
3	50	1	1	66.19%
4	100	1	1	66.46%
5	100	1	2	67.27%
6	100	2	2	66.34%
7	100	3	3	67.00%
8	100	4	4	65.03%
9	200	1	1	66.70%
10	500	1	1	66.37%
11	500	2	2	66.07%
12	500	3	3	67.00%
13	500	4	4	65.88%

Table B.2: F_1 values for different configurations (see B.1). Trained and tested on our own split of the Semantic3D dataset.

	Terrain		Vegetation		Build-	Hard	Arte-		Weight.
Conf. #	**Man-m.**	**Nat.**	**High**	**Low**	**ings**	**Scape**	**facts**	**Cars**	**Avg.**
1	0.861	0.109	0.756	0.270	0.669	0.102	0.349	0.157	0.683
2	0.872	0.109	0.752	0.277	0.677	0.036	0.388	0.172	0.686
3	0.872	0.106	0.756	0.287	0.689	0.022	0.391	0.174	0.691
4	0.880	0.107	0.752	0.289	0.688	0.024	0.410	0.168	0.692
5	0.893	0.127	0.758	0.290	0.686	0.025	0.411	0.167	0.697
6	0.875	0.106	0.760	0.278	0.681	0.030	0.420	0.164	0.691
7	0.898	0.131	0.757	0.277	0.678	0.031	0.407	0.161	0.696
8	0.864	0.103	0.751	0.273	0.673	0.027	0.411	0.166	0.683
9	0.881	0.107	0.752	0.287	0.691	0.030	0.419	0.175	0.693
10	0.878	0.107	0.753	0.284	0.687	0.027	0.418	0.172	0.692
11	0.875	0.107	0.754	0.281	0.680	0.026	0.422	0.167	0.689
12	0.898	0.131	0.757	0.277	0.678	0.031	0.407	0.161	0.696
13	0.877	0.108	0.752	0.278	0.680	0.024	0.415	0.164	0.688

Appendix C

Parameter Optimization: Precision and Recall on Different Scales

Table C.1: Precision values for our random forest classifier using different scales for feature calculation.

	Class Precision Score								Weight.
Red.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Avg.**
0	**0.964**	0.278	0.776	**0.244**	**0.840**	0.085	0.278	0.076	0.765
1	0.954	0.294	0.764	0.240	0.801	0.151	**0.284**	0.074	0.750
2	0.958	**0.298**	0.782	0.242	0.805	0.124	0.252	0.082	0.757
5	0.958	0.291	0.793	0.238	0.785	0.162	0.172	0.079	0.754
10	0.959	0.260	0.816	0.224	0.764	**0.250**	0.149	0.072	0.756
20	0.960	0.254	**0.838**	0.194	0.787	0.221	0.136	0.108	**0.767**
40	0.937	0.212	0.812	0.221	0.815	0.118	0.115	0.094	0.757
80	0.899	0.223	0.796	0.220	0.773	0.083	0.116	0.113	0.730
160	0.831	0.032	0.729	0.211	0.714	0.088	0.153	**0.123**	0.670
2, 10	**0.961**	0.311	0.854	**0.246**	0.773	0.151	0.238	0.076	0.772
2, 20	0.952	**0.360**	**0.883**	0.243	0.761	**0.185**	0.218	0.074	0.777
2, 80	0.939	0.328	0.875	0.228	**0.812**	0.128	**0.251**	**0.149**	**0.784**
2, 10, 80	0.936	0.309	0.880	0.252	0.863	0.212	**0.266**	**0.132**	**0.802**
2, 20, 80	0.934	**0.371**	**0.890**	0.244	0.858	0.164	**0.266**	0.119	**0.802**
2, 8, 32	0.957	0.301	0.862	**0.291**	**0.879**	0.178	0.243	0.113	0.800
5, 20, 80	0.935	0.341	0.888	0.248	0.857	0.201	0.214	0.128	**0.802**
10, 20, 40	**0.958**	0.313	0.874	0.230	0.835	**0.220**	0.166	0.124	0.795

Table C.2: Recall values for our random forest classifier using different scales for feature calculation.

	Class Recall Score								Weight.
Red.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Avg.**
0	0.928	0.495	**0.740**	0.532	0.441	0.006	**0.677**	0.514	**0.665**
1	**0.940**	0.430	0.715	**0.562**	0.415	0.027	0.594	0.518	0.652
2	0.920	0.580	0.709	0.558	0.443	0.029	0.648	0.501	0.657
5	0.906	0.590	0.699	0.542	0.460	0.031	0.577	0.424	0.652
10	0.877	0.578	0.692	0.484	0.471	0.115	0.477	0.419	0.643
20	0.800	**0.711**	0.687	0.392	**0.644**	0.102	0.311	0.480	0.664
40	0.775	0.613	0.724	0.162	0.626	**0.159**	0.301	**0.639**	0.652
80	0.890	0.366	0.694	0.228	0.605	0.063	0.377	0.609	0.663
160	0.713	0.099	0.651	0.332	0.544	0.017	0.392	0.618	0.583
2, 10	0.889	**0.704**	0.692	0.494	0.494	0.040	**0.662**	**0.577**	0.658
2, 20	0.915	0.669	0.694	0.462	0.533	0.043	0.638	0.534	0.674
2, 80	**0.937**	0.428	**0.718**	**0.565**	**0.610**	**0.059**	0.648	0.459	**0.708**
2, 10, 80	0.916	0.484	0.741	0.535	0.639	0.118	0.679	0.536	0.720
2, 20, 80	**0.927**	0.509	**0.747**	0.469	0.650	0.097	0.662	**0.551**	**0.724**
2, 8, 32	0.886	0.650	0.702	**0.569**	0.622	**0.142**	**0.687**	0.526	0.695
5, 20, 80	0.917	0.488	0.735	0.538	**0.656**	0.090	0.611	0.478	0.721
10, 20, 40	0.880	**0.659**	0.722	0.457	0.654	0.119	0.484	0.481	0.705

Appendix D

Detailed Results of Comparing Classifiers

D.1 Oakland

Table D.1: Comparison of precision scores on the Oakland dataset.

	Class Precision Score				
Classifier	**1**	**2**	**3**	**4**	**5**
Original RF	0.959	0.091	0.236	0.972	0.846
Our RF	0.953	0.101	**0.376**	**0.999**	0.893
Our SVM_{RBF}	0.767	0.020	0.053	0.998	0.769
Our SVM_{Linear}	0.805	0.015	0.061	0.997	0.715
KPConv	**0.964**	**0.104**	0.327	0.998	**0.933**

Table D.2: Comparison of recall scores on the Oakland dataset.

	Class Recall Score				
Classifier	**1**	**2**	**3**	**4**	**5**
Original RF	0.809	0.862	**0.798**	0.985	0.672
Our RF	0.922	0.894	0.751	**0.994**	0.696
Our SVM_{RBF}	0.776	0.764	0.719	0.809	0.289
Our SVM_{Linear}	0.816	0.820	0.427	0.809	0.220
KPConv	**0.968**	**0.900**	0.623	0.985	**0.720**

Table D.3: Confusion matrix - Our RF on Oakland.

Act. \ Pred.	**1**	**2**	**3**	**4**	**5**
1	**246442**	9366	1800	1128	8589
2	288	**3393**	21	2	90
3	1434	249	**5954**	23	273
4	2896	2454	20	**928480**	296
5	7464	18141	8059	102	**77346**

Table D.4: Confusion matrix - KPConv on Oakland.

Act. \ Pred.	**1**	**2**	**3**	**4**	**5**
1	**258644**	3234	1281	661	3505
2	334	**3415**	22	23	0
3	1248	132	**4939**	56	1558
4	3658	9334	154	**920312**	688
5	4465	16852	8727	1048	**80020**

D.2 Paris-rue-Madame

Table D.5: Comparison of precision scores on the Paris-rue-Madame dataset.

	Class Precision Score					
Classifier	**1**	**2**	**3**	**4**	**5**	**6**
Original RF	0.962	0.964	0.755	**0.019**	0.123	0.055
Our RF	**0.984**	**0.994**	0.816	0.000	0.291	0.200
Our SVM_{RBF}	0.975	0.984	0.895	0.000	0.087	0.017
Our SVM_{Linear}	0.973	0.989	0.891	0.000	0.032	0.036
KPConv	0.981	**0.994**	**0.989**	0.000	**0.884**	**0.996**

Table D.6: Comparison of recall scores on the Paris-rue-Madame dataset.

	Class Recall Score					
Classifier	**1**	**2**	**3**	**4**	**5**	**6**
Original RF	0.957	0.902	0.606	**0.575**	**0.639**	**0.974**
Our RF	0.982	0.937	**0.995**	0.000	0.154	0.635
Our SVM_{RBF}	0.972	0.871	0.827	0.011	0.188	0.752
Our SVM_{Linear}	0.981	0.851	0.839	0.000	0.108	0.860
KPConv	**0.998**	**0.982**	0.957	0.000	0.625	0.623

Table D.7: Confusion matrix - Our RF on Paris-Rue-Madame.

Act. \ Pred.	1	2	3	4	5	6
1	**5115398**	15492	70100	51	1909	6068
2	79831	**3457036**	146914	659	4142	2654
3	149	4076	**1039714**	221	401	0
4	960	7	4681	**0**	0	744
5	803	681	12028	221	**2642**	747
6	1463	3	0	0	0	**2551**

Table D.8: Confusion matrix - KPConv on Paris-Rue-Madame.

Act. \ Pred.	1	2	3	4	5	6
1	**5199160**	9858	0	0	0	0
2	59019	**3624301**	7090	0	816	10
3	34156	10711	**999197**	0	497	0
4	5942	1	449	**0**	0	0
5	1971	635	3807	0	**10709**	0
6	1390	35	0	0	89	**2503**

D.3 Sim2Real

Table D.9: Comparison of precision scores when trained on Sim2Real.

	Class Precision Score					
Classifier	**1**	**2**	**3**	**4**	**5**	**6**
Our RF	**1.000**	0.784	0.876	0.893	0.959	0.827
Our SVM_{RBF}	0.998	0.536	**0.964**	0.157	0.697	0.922
Our SVM_{Linear}	0.994	0.232	0.955	0.067	0.783	**0.938**
KPConv	0.999	**0.871**	0.800	**0.929**	**0.967**	0.909

Table D.10: Comparison of recall scores when trained on Sim2Real.

	Class Recall Score					
Classifier	**1**	**2**	**3**	**4**	**5**	**6**
Our RF	0.973	0.762	0.961	0.304	0.650	**0.986**
Our SVM_{RBF}	0.979	**0.867**	0.819	0.499	0.901	0.860
Our SVM_{Linear}	0.931	0.864	0.837	**0.586**	0.895	0.594
KPConv	**0.981**	0.230	**0.999**	0.233	**0.925**	0.845

Table D.11: Confusion matrix - Our RF on Sim2Real.

Act. \ Pred.	1	2	3	4	5	6
1	**6212347**	26761	19840	5	50	122845
2	169	**159496**	15309	29	43	34301
3	6	2697	**1752082**	633	1771	66512
4	268	4193	11463	**19649**	18104	10898
5	5	3659	189184	1575	**490993**	69451
6	246	6568	12681	107	919	**1457669**

Table D.12: Confusion matrix - KPConv on Sim2Real.

Act. \ Pred.	1	2	3	4	5	6
1	**6260474**	4611	93398	11	11459	11895
2	4266	**48177**	50699	203	1501	104501
3	1064	240	**1821503**	425	342	127
4	6	985	34595	**15059**	10464	3466
5	71	1247	50105	410	**698364**	4670
6	2313	23	226584	103	66	**1249101**

D.4 Semantic3D Challenge

Results with color information of the authors of KPConv can be found at `http://www.semantic3d.net/view_method_detail.php?method=KP-FCNN`.

Table D.13: Comparison of precision scores on Semantic3D.

	Class Precision Score							
Classifier	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Our RF	0.977	0.873	0.646	0.506	**0.961**	0.390	0.224	0.478
KPConv	**0.989**	**0.962**	**0.821**	**0.678**	0.950	**0.688**	**0.712**	**0.944**

Table D.14: Comparison of recall scores on Semantic3D.

	Class Recall Score							
Classifier	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Our RF	0.905	**0.955**	0.882	**0.582**	0.893	0.264	0.679	0.703
KPConv	**0.993**	0.938	**0.989**	0.453	**0.992**	**0.373**	**0.807**	**0.866**

Table D.15: Confusion matrix - Our RF on Semantic3D.

Act. \ Pred.	1	2	3	4	5	6	7	8
1	**12951866**	1150287	18418	103437	19675	19349	10003	4447
2	197760	**10216373**	2965	194123	6868	51665	1746	23246
3	9	865	**4124839**	304765	145582	55702	36093	8016
4	6040	194580	667482	**1758590**	50533	103550	51632	189487
5	86046	124814	1402283	651191	**27591613**	710196	238779	106284
6	7658	9592	151616	347803	784780	**629731**	190675	262690
7	572	502	11883	7054	19858	14962	**158757**	20251
8	1434	995	1215	105935	92052	30249	21030	**599032**

Table D.16: Confusion matrix - KPConv on Semantic3D.

Act. \ Pred.	1	2	3	4	5	6	7	8
1	**14222419**	27827	87	1163	38542	16977	7369	3125
2	77759	**10033009**	5182	553962	20681	4013	61	79
3	0	1514	**4625609**	10043	31319	5029	2292	65
4	31737	338222	984133	**1369104**	81573	179988	31312	582
5	27396	10011	4366	11145	**30672552**	183650	1939	147
6	24007	17224	14011	72032	1305063	**888420**	29405	34383
7	1609	199	1627	278	29699	11628	**188768**	31
8	2064	13	0	623	105499	2003	3799	**737941**

Bibliography

[1] Mario Amrehn, Firas Mualla, Elli Angelopoulou, Stefan Steidl, and Andreas Maier. The random forest classifier in Weka: Discussion and new developments for imbalanced data. *arXiv preprint arXiv:1812.08102*, 2018.

[2] Karen Anderson, Steven Hancock, Mathias Disney, and Kevin J Gaston. Is waveform worth it? A comparison of lidar approaches for vegetation and landscape characterization. *Remote Sensing in Ecology and Conservation*, 2(1):5–15, 2016.

[3] S. Bechtold et al. Arff stable - Weka wiki. `https://waikato.github.io/weka-wiki/formats_and_processing/arff_stable`, 2019. Accessed: 2020-02-01.

[4] S. Bechtold et al. Manual - giscience/helios wiki. `https://github.com/GIScience/helios/wiki/Manual`, 2019. Accessed: 2020-02-01.

[5] S. Bechtold and B. Höfle. HELIOS: A multi-purpose lidar simulation framework for research, planning and training of laser scanning operations with airborne, ground-based mobile and stationary platforms. *ISPRS Annals of Photogrammetry, Remote Sensing & Spatial Information Sciences*, 3(3):161–168, 2016.

[6] Yizhak Ben-Shabat, Michael Lindenbaum, and Anath Fischer. 3DmFV: Three-dimensional point cloud classification in real-time using convolutional neural networks. *IEEE Robotics and Automation Letters*, 3(4):3145–3152, 2018.

[7] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[8] Paul Berner. Orientation, rotation, velocity, and acceleration and the srm. *SEDRIS Organization, ISO/IEC JTC*, 1, 2007.

[9] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor fusion IV: control paradigms and data structures*, volume 1611, pages 586–606. International Society for Optics and Photonics, 1992.

[10] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[11] Dorit Borrmann. Multi-modal 3d mapping - combining 3d point clouds with thermal and color information. In *Schriftenreihe Würzburger Forschungsberichte in Robotik und Telematik, Band 14*. Universität Würzburg, 2018, 2018.

[12] Dorit Borrmann, Jan Elseberg, Prashant Narayan KC, and Andreas Nüchter. Ein punkt pro kubikmeter – präzise registrierung von terrestrischen laserscans mit scanmatching. *Photogrammetrie Laserscanning Optische 3D-Messtechnik, Beiträge der Oldenburger 3D-Tage*, 2012.

[13] Dorit Borrmann, Jan Elseberg, and Andreas Nüchter. Thermal 3d mapping of building façades. In *Intelligent autonomous systems 12*, pages 173–182. Springer, 2013.

[14] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

[15] Alexandre Boulch, Bertrand Le Saux, and Nicolas Audebert. Unstructured point cloud semantic labeling using deep segmentation networks. *3DOR*, 2:7, 2017.

[16] Paul Bourke. Object files. `http://paulbourke.net/dataformats/obj`. Accessed: 2020-02-01.

[17] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[18] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[19] Allan Brito. *Blender Quick Start Guide: 3D Modeling, Animation, and Render with Eevee in Blender 2.8*. Packt Publishing Ltd, 2018.

[20] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[21] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[22] Chao Chen, Andy Liaw, Leo Breiman, et al. Using random forest to learn imbalanced data. *University of California, Berkeley*, 110(1-12):24, 2004.

[23] Nancy Chinchor. MUC-4 evaluation metrics. In *Proceedings of the 4th conference on Message understanding*, pages 22–29. Association for Computational Linguistics, 1992.

[24] Dan Ciresan, Alessandro Giusti, Luca M Gambardella, and Jürgen Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In *Advances in neural information processing systems*, pages 2843–2851, 2012.

[25] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(Aug):2493–2537, 2011.

[26] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[27] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 764–773, 2017.

[28] Mark De Deuge, Alastair Quadros, Calvin Hung, and Bertrand Douillard. Unsupervised feature learning for classification of outdoor 3d scans. In *Australasian Conference on Robitics and Automation*, volume 2, page 1, 2013.

[29] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[30] Jérôme Demantké, Clément Mallet, Nicolas David, and Bruno Vallet. Dimensionality based scale selection in 3d lidar point clouds. 2011.

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[32] Rossen Dimov, Michael Feld, Dr Michael Kipp, Dr Alassane Ndiaye, and Dr Dominik Heckmann. Weka: Practical machine learning tools and techniques with java implementations. *AI Tools SeminarUniversity of Saarland, WS*, 6(07), 2007.

[33] Chris Drummond, Robert C Holte, et al. C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *Workshop on learning from imbalanced datasets II*, volume 11, pages 1–8. Citeseer, 2003.

[34] Emil Dumic, Anamaria Bjelopera, and Andreas Nüchter. Projection based dynamic point cloud compression using 3DTK toolkit and H.265/HEVC. In *2019 2nd International Colloquium on Smart Grid Metrology (SMAGRIMET)*, pages 1–4. IEEE, 2019.

[35] Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. Full wave analysis in 3d laser scans for vegetation detection in urban environments. In *2011 XXIII International Symposium on Information, Communication and Automation Technologies*, pages 1–7. IEEE, 2011.

[36] Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. One billion points in the cloud - an octree for efficient processing of 3d laser scans. *ISPRS Journal of Photogrammetry and Remote Sensing*, 76:76–88, 2013.

[37] Mark Everingham, SM Ali Eslami, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes challenge: A retrospective. *International journal of computer vision*, 111(1):98–136, 2015.

[38] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[39] Sagi Filin and Norbert Pfeifer. Neighborhood systems for airborne laser data. *Photogrammetric Engineering & Remote Sensing*, 71(6):743–755, 2005.

[40] J Friedman, T Hastie, and R Tibshirani. The elements of statistical learning: Springer series in statistics springer, 2001.

[41] Simone Frintrop. *Robuste Roboterlokalisierung mit omnidirektionaler Bildsensorik*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2001.

[42] Daniel Girardeau-Montaut. Cloudcompare – point cloud processing workshop, 2016.

[43] Timo Hackel, Nikolay Savinov, Lubor Ladicky, Jan D Wegner, Konrad Schindler, and Marc Pollefeys. Semantic3d. net: A new large-scale point cloud classification benchmark. *arXiv preprint arXiv:1704.03847*, 2017.

[44] Timo Hackel, Jan D Wegner, Nikolay Savinov, Lubor Ladicky, Konrad Schindler, and Marc Pollefeys. Large-scale supervised learning for 3d point cloud labeling: Semantic3d.net. *Photogrammetric Engineering & Remote Sensing*, 84(5):297–308, 2018.

[45] Timo Hackel, Jan D Wegner, and Konrad Schindler. Fast semantic segmentation of 3d point clouds with strongly varying density. *ISPRS annals of the photogrammetry, remote sensing and spatial information sciences*, 3:177–184, 2016.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[47] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[48] Ekbert Hering and Gert Schönfelder. *Sensoren in Wissenschaft und Technik: Funktionsweise und Einsatzgebiete*. Springer, 2018.

[49] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[50] Badr Hssina, Abdelkarim Merbouha, Hanane Ezzikouri, and Mohammed Erritali. A comparative study of decision tree ID3 and C4. 5. *International Journal of Advanced Computer Science and Applications*, 4(2):13–19, 2014.

[51] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification, 2003.

[52] Qingyong Hu, Bo Yang, Linhai Xie, Stefano Rosa, Yulan Guo, Zhihua Wang, Niki Trigoni, and Andrew Markham. RandLA-Net: Efficient semantic segmentation of large-scale point clouds. *arXiv preprint:1911.11236*, 2019.

[53] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2012.

[54] B Jutzi and H Gross. Nearest neighbour classification on laser point clouds to gain object structures from buildings. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38(Part 1):4–7, 2009.

[55] Kari Karhunen. *Über lineare Methoden in der Wahrscheinlichkeitsrechnung*, volume 37. Sana, 1947.

[56] Thomas P Kersten, Felix Tschirschwitz, Simon Deggim, and Maren Lindstaedt. Virtual reality for cultural heritage monuments–from 3d data recording to immersive visualisation. In *Euro-Mediterranean Conference*, pages 74–83. Springer, 2018.

[57] Hans-Joachim Kowalsky. *Lineare algebra*. Walter de Gruyter, 2013.

[58] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[59] Miroslav Kubat, Stan Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. In *Icml*, volume 97, pages 179–186. Nashville, USA, 1997.

[60] Shir Meir Lador. What metrics should be used for evaluating a model on an imbalanced data set? (precision + recall or roc=tpr+fpr). `https://towardsdatascience.com/what-metrics-should-we-use-on-imbalanced-data-set-precision-recall-roc-e2e79252aeba`, 2017. Accessed: 2020-02-11.

[61] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. pages 55–63. IEEE, 2010.

[62] Jean-François Lalonde, Nicolas Vandapel, Daniel F Huber, and Martial Hebert. Natural terrain classification using three-dimensional ladar data for ground robot mobility. *Journal of field robotics*, 23(10):839–861, 2006.

[63] Loic Landrieu and Martin Simonovsky. Large-scale point cloud semantic segmentation with superpoint graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4558–4567, 2018.

[64] Felix Järemo Lawin, Martin Danelljan, Patrik Tosteberg, Goutam Bhat, Fahad Shahbaz Khan, and Michael Felsberg. Deep projective 3d semantic segmentation. In *International Conference on Computer Analysis of Images and Patterns*, pages 95–107. Springer, 2017.

[65] Impyeong Lee and Toni Schenk. Perceptual organization of 3d surface points. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 34(3/A):193–198, 2002.

[66] Jiaxin Li, Ben M Chen, and Gim Hee Lee. SO-Net: Self-organizing network for point cloud analysis. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9397–9406, 2018.

[67] Velodyne Lidar. HDL-64E. `https://velodynelidar.com/products/hdl-64e/`, 2020. Accessed: 2020-03-01.

[68] Charles X Ling and Chenghui Li. Data mining for direct marketing: Problems and solutions. In *Kdd*, volume 98, pages 73–79, 1998.

[69] Michel Loève. Probability theory 1. *New York*, 1963.

[70] Hans-Gerd Maas and George Vosselman. Two algorithms for extracting building models from raw laser altimetry data. *ISPRS Journal of photogrammetry and remote sensing*, 54(2-3):153–163, 1999.

[71] Clément Mallet, Frédéric Bretar, Michel Roux, Uwe Soergel, and Christian Heipke. Relevance assessment of full-waveform lidar data for urban area classification. *ISPRS journal of photogrammetry and remote sensing*, 66(6):S71–S84, 2011.

[72] Jonathan Masci, Davide Boscaini, Michael Bronstein, and Pierre Vandergheynst. Geodesic convolutional neural networks on riemannian manifolds. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 37–45, 2015.

[73] Daniel Maturana and Sebastian Scherer. VoxNet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928. IEEE, 2015.

[74] Grégoire Montavon, Matthias Rupp, Vivekanand Gobre, Alvaro Vazquez-Mayagoitia, Katja Hansen, Alexandre Tkatchenko, Klaus-Robert Müller, and O Anatole Von Lilienfeld. Machine learning of molecular electronic properties in chemical compound space. *New Journal of Physics*, 15(9):095003, 2013.

[75] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15, 2018.

[76] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5115–5124, 2017.

[77] Daniel Munoz, J Andrew Bagnell, Nicolas Vandapel, and Martial Hebert. Contextual classification with functional max-margin markov networks. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 975–982. IEEE, 2009.

[78] Hiroshi Murase and Shree K Nayar. Visual learning and recognition of 3-d objects from appearance. *International journal of computer vision*, 14(1):5–24, 1995.

[79] Andreas Nüchter, Dorit Borrmann, and Johannes Schauer. 3DTK - the 3d toolkit. `http://threedtk.de`, 2019. Accessed: 2020-02-01.

[80] Andreas Nüchter and K Lingemann. 3DTK - the 3d toolkit. `https://robotik.informatik.uni-wuerzburg.de/telematics/download/IAS-tutorial-01.pdf`, 2014. Accessed: 2020-02-01.

[81] Josh Patterson and Adam Gibson. *Getting started with deep learning.* O'Reilly Media, Inc., 2018.

[82] Karl Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[83] John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[84] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. Numerical recipes in c, 1988.

[85] Philipp Probst. *Hyperparameters, tuning and meta-learning for random forest and other machine learning algorithms.* PhD thesis, lmu, 2019.

[86] Philipp Probst and Anne-Laure Boulesteix. To tune or not to tune the number of trees in random forest. *The Journal of Machine Learning Research*, 18(1):6673–6690, 2017.

[87] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. PointNet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[88] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. PointNet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in neural information processing systems*, pages 5099–5108, 2017.

[89] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[90] Diane Ramey, Linda Rose, and Lisa Tyerman. Mtl material format (lightwave, obj). `http://paulbourke.net/dataformats/mtl`, 1995. Accessed: 2020-02-01.

[91] Riegl. Riegl VZ-400 datasheet. `http://www.riegl.com/uploads/tx_pxpriegldownloads/10_DataSheet_VZ-400_2017-06-14.pdf`, 2017. Accessed: 2020-03-01.

[92] Brian D Ripley. *Pattern recognition and neural networks.* Cambridge university press, 2007.

[93] Jürgen Rossmann, Michael Schluse, Arno Bücken, and Petra Krahwinkler. Using airborne laser-scanner-data in forestry management: A novel approach to single tree delineation. In *Proceedings of the ISPRS Workshop on Laser Scanning*, pages 350–354, 2007.

[94] Xavier Roynard, Jean-Emmanuel Deschaud, and François Goulette. Classification of point cloud scenes with multiscale voxel deep network. *arXiv preprint arXiv:1804.03583*, 2018.

[95] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. ImageNet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[96] Bernhard Schölkopf, Alexander J Smola, Francis Bach, et al. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[97] Mark R Segal. Machine learning benchmarks and random forest regression. 2004.

[98] Heidi Seibold, Christoph Bernau, Anne-Laure Boulesteix, and Riccardo De Bin. On the choice and influence of the number of boosting steps for high-dimensional linear cox-models. *Computational Statistics*, 33(3):1195–1215, 2018.

[99] Andrés Serna, Beatriz Marcotegui, François Goulette, and Jean-Emmanuel Deschaud. Paris-rue-madame database: a 3d mobile laser scanner dataset for benchmarking urban detection, segmentation and classification methods. 2014.

[100] Lindsay I Smith. A tutorial on principal components analysis. Technical report, 2002.

[101] Shuran Song, Samuel P Lichtenberg, and Jianxiong Xiao. SUN RGB-D: A rgb-d scene understanding benchmark suite. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 567–576, 2015.

[102] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. Multi-view convolutional neural networks for 3d shape recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 945–953, 2015.

[103] Maxim Tatarchenko, Jaesik Park, Vladlen Koltun, and Qian-Yi Zhou. Tangent convolutions for dense prediction in 3d. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3887–3896, 2018.

[104] Hugues Thomas, François Goulette, Jean-Emmanuel Deschaud, and Beatriz Marcotegui. Semantic classification of 3d point clouds with multiscale spherical neighborhoods. In *2018 International Conference on 3D Vision (3DV)*, pages 390–398. IEEE, 2018.

[105] Hugues Thomas, Charles R Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, François Goulette, and Leonidas J Guibas. KPConv: Flexible and deformable convolution for point clouds. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 6411–6420, 2019.

[106] Hugues Thomas, Charles R Qi, Jean-Emmanuel Deschaud, Beatriz Marcotegui, François Goulette, and Leonidas J Guibas. Supplementary material for KPConv: Flexible and deformable convolution for point clouds. `http://openaccess.thecvf.com/content_ICCV_2019/supplemental/Thomas_KPConv_Flexible_and_ICCV_2019_supplemental.pdf`, 2019. Accessed: 2020-04-01.

[107] Alexander Toshev, Philippos Mordohai, and Ben Taskar. Detecting and parsing architecture at city scale from range data. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 398–405. IEEE, 2010.

[108] Matthew Turk and Alex Pentland. Face recognition using eigenfaces. In *Proceedings. 1991 IEEE computer society conference on computer vision and pattern recognition*, pages 586–587, 1991.

[109] Bruno Vallet, Mathieu Brédif, Andrés Serna, Beatriz Marcotegui, and Nicolas Paparoditis. Terramobilita/iqmulus urban point cloud analysis benchmark. *Computers & Graphics*, 49:126–133, 2015.

[110] Vladimir Vapnik. Estimation of dependences based on empirical data berlin, 1982.

[111] William N Venables and Brian D Ripley. *Modern applied statistics with S-PLUS*. Springer Science & Business Media, 2013.

[112] Walber. Precision and recall - wikimedia commons. `https://commons.wikimedia.org/wiki/File:Precisionrecall.svg`, 2014. CC-BY-SA-4.0. Accessed: 2020-03-01.

[113] Andrew R Webb. *Statistical pattern recognition*. John Wiley & Sons, 2003.

[114] Martin Weinmann, Boris Jutzi, and Clément Mallet. Feature relevance assessment for the semantic interpretation of 3d point cloud data. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 5(W2):1, 2013.

[115] Martin Weinmann, Boris Jutzi, and Clément Mallet. Semantic 3d scene interpretation: A framework combining optimal neighborhood size selection with relevant features. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2(3):181, 2014.

[116] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[117] Ian H Witten, Eibe Frank, Leonard E Trigg, Mark A Hall, Geoffrey Holmes, and Sally Jo Cunningham. Weka: Practical machine learning tools and techniques with java implementations. 1999.

[118] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3D ShapeNets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[119] Yifan Xu, Tianqi Fan, Mingye Xu, Long Zeng, and Yu Qiao. SpiderCNN: Deep learning on point sets with parameterized convolutional filters. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 87–102, 2018.

[120] Li Yi, Hao Su, Xingwen Guo, and Leonidas J Guibas. SyncSpecCNN: Synchronized spectral cnn for 3d shape segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2282–2290, 2017.

Proclamation

Hereby I confirm that I wrote this thesis independently and that I have not made use of any other resources or means than those indicated.

Würzburg, May 2020