University of Stavanger

THE FACULTY OF TECHNICAL AND NATURAL SCIENCES

MASTER'S THESIS

Study programme / specialisation: Master's in engineering / Data science	Spring semester 2023 Open
Author(s): Yohannes Dawit Kassaye and Sigurd Grøvdal Hansen	
Faculty supervisor: Ferhat Özgur Catak Supervisor(s): Hassan Sartaj and Shaukat Ali	
Thesis title: Developing and Testing Digital Twins for Vehicle Collision Prediction: A Machine Learning and Genetic Search Algorithm Approach	
Credits (ECTS): 60 ($30 \cdot 2$)	
Keywords: Digital twins, machine learning, genetic algorithm, self-driving cars	Pages: 86 + appendix: 1 page Stavanger 15. june 2023

Abstract

This thesis focuses on developing a digital twin which can predict and avoid collisions. The digital twin does this by using different machine learning models that are trained on data from the SVL Simulator. By harnessing the power of machine learning, the digital twin demonstrates promising abilities in collision prediction and prevention. Additionally, a genetic search algorithm is developed to generate specialized testing data, enabling comprehensive evaluation of the digital twin's performance.

The central contribution of this research lies in exploring the viability of utilizing test data that is generated by a genetic search algorithm to evaluate the performance of the digital twin. By employing the genetic search algorithm to generate data resembling real collision scenarios, classified as collisions, an interesting evaluation framework is established. Through the evaluation process, which involves analyzing the number of accurately classified collisions by the digital twin, insights are gained into the model's effectiveness in predicting collisions.

This contributes to the ongoing efforts in enhancing the accuracy of collision prediction systems, ultimately leading to improved safety measures in autonomous driving and intelligent transportation systems.

Acknowledgements

We would like to thank Hassan Sartaj and Shaukat Ali for guiding us through the development and the writing of this thesis. They also gave us good advice on what to research and ideas.

Contents

List of Figures **vii**

List of Tables **x**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem definition 2

1.3 Research questions 2

1.4 Outline 2

2 Background **4**

2.1 Digital twins 4

2.2 Self-driving cars 6

2.3 SVL Simulator 7

2.3.1 WISE 9

2.3.2 OSSDC-Sim 10

2.3.3 Python API 11

2.4 Deep Scenario 16

2.4.1 Scenarios 16

2.4.2 Features 17

2.4.3 Deep scenario XML-files 18

3 Solution Approach **20**

3.1 Generated collision data 20

3.2 Machine learning for collision prediction 22

3.2.1 Introduction to machine learning 22

3.2.2 Data preprocessing and feature engineering 23

3.2.3 Machine learning models for collision prediction . . . 24

3.2.4 Evaluation and performance metrics 27

3.3 Genetic algorithm 30

3.3.1 Simple implementation example 31

3.3.2 Variations of the algorithm 32

4 Implementation **39**

4.1 Generating the self generated data 40

4.2 Feature engineering 40

4.2.1 Deep Scenario 41

4.2.2 Self-generated data 45

4.3 Implementation of the features 48

4.3.1 Distance to obstacle 48

4.3.2 Time to collision 53

4.3.3 Jerk 56

4.4 Implementation of the model 56

4.4.1 Preprocessing 57

4.4.2 Machine Learning Models 58

4.5 Using the simulator 59

4.5.1 Getting started 60

4.5.2 Creation of the twins 60

4.6 Implementation of the genetic algorithm 64

4.6.1 Population 64

4.6.2 Fitness 65

4.6.3 Selection 66

4.6.4 Crossover 66

4.6.5 Mutation 66

4.6.6 Running the algorithm 67

5 Results and discussion **68**

5.1 Digital twin 68

5.2 Genetic algorithm 75

5.3 Answering the research questions 79

6 Conclusion **81**

Bibliography **86**

A Source Code **87**

List of Figures

2.1 Communication between a physical and a digital twin.[1] . . . 5

2.2 Before starting a simulation with the Python API with the original SVL Simulator. 8

2.3 Before starting a simulation with the Python API with the original SVL Simulator. 10

2.4 Launching the simulator. 11

2.5 An overview over how the simulator and the API communicates. 12

2.6 Explanation about how a LiDAR calculates the distance to an obstacle.[2] 14

2.7 A normal view from within the simulator. Here one can see the ego vehicle and an NPC vehicle. 15

2.8 How the LiDAR sees the same image as in figure 2.7. Notice how the sensor captures the back of the car and the closest tree. 15

3.1 How the different parts of a genetic algorithm follow each other. In this case the stopping criteria is either that the population is fit enough or that the generation limit is reached. 31

LIST OF FIGURES

3.2 Showing the distribution of the individuals in a roulette wheel. 34

3.3 Distribution of which rank is picked when different number of participants are used. 35

3.4 Three different crossover functions, single point, uniform and average. 37

3.5 An individual gets two of its bits flipped. 37

4.1 A simple overview of how the different parts are tied together. 39

4.2 A top down view of the point cloud, the ego vehicles is at the center. Each "circle" is from a different laser looking 1.3 degrees further up. The direction of the ego vehicle is to the right. 51

4.3 What the LiDAR sees when the same NPC vehicle is located at different distances from the ego vehicle. The bottom part of each section is the same point cloud from a top down view. 53

4.4 The plot generated after running scenario 1 with the XGB classifier as the brains of the digital twin. In this scenario, the ego vehicle stopped right before colliding with another vehicle. The scenario is explained in section 5.1. 63

4.5 The ego vehicle manages to stop in time when the digital twin tells the physical twin to apply the brakes. 64

5.1 Comparing the prediction collision with the actual collisions. Here each model has been trained with data from the Deep Scenario dataset. 1 represents a collision prediction and 0 for not a collision. 71

5.2 Comparing the prediction collision with the actual collisions. Here each model has been trained with the self-generated data. 1 represents a collision prediction and 0 for not a collision. 73

5.3 Evolution of the fitness, for the most fit individual and the population average. Note that the y-axis for the two lines are different. 76

5.4 How the top 5 individuals from the genetic algorithm can look like. The features about angular velocity have been removed in the figure to make it more readable. 77

List of Tables

2.1 A peek at the data stored in the csv-files. 17

3.1 The correlations for the multiple features and from both datasets regarding collision. The feature type with an average below 0.1 is removed from the table. 21

3.2 Raw self-generated data. The processed data can be seen in table 4.1. 22

4.1 The processed data after being made into usable data. When comparing with table 3.2, notice how the time steps 15.0 and 15.5 has been removed as they are actual collision rows and that the four rows above now is a collision. 47

5.1 Confusion matrix for each model that is either trained with data from Deep Scenario of the self-generated one. The values are from predicting with the test data from each dataset. Top left represents the true negatives, top right represents the false positives, bottom left represents the false negatives and the bottom right represents the true positives. 69

5.2 The input parameters to the genetic algorithm. 76

LIST OF TABLES

5.3 The matches from the genetic algorithm with actual collision data. 78

5.4 Predictions that are a collision from the 100 best individuals from five different runs of the genetic algorithm, top 20 from each. 78

Chapter 1

Introduction

This chapter initiates the thesis by presenting the motivation behind it. Subsequently, a problem statement will be introduced, accompanied by research questions. Finally, the thesis will be outlined, providing a clear overview of its structure and content.

1.1 Motivation

The concept of digital twins has gained significant attention and relevance in recent times. This thesis aims to delve deeper into the technology and explore the process of creating a digital twin. The authors possess a genuine interest in automobiles and have been actively involved with ION Racing in the past. ION Racing is an organization focused on developing a formula student car to participate in prestigious competitions like Formula Student. Notably, Formula Student encompasses a branch dedicated to self-driving cars, which adds an intriguing dimension to this thesis work. By delving into the topic of digital twins, this thesis has the potential to contribute to the advancement of autonomous cars—a pivotal aspect of the evolving future global landscape.

1.2 Problem definition

In the rapidly advancing field of self-driving cars, reliable collision prediction systems are essential for ensuring passenger and pedestrian safety. The research focuses on developing a collision predicting Digital Twin using machine learning, and use genetic search algorithm to generate comprehensive test data, enabling thorough evaluation and validation of the Digital Twin's performance.

1.3 Research questions

1. *RQ1*: How can machine learning be utilized to develop a digital twin that has the capabilities to detect and avoid collisions?
2. *RQ2*: Given that RQ1 has a viable solution, how can a genetic search algorithm be used to create testing data for a digital twin?
3. *RQ3*: Is a genetic search based test dataset a viable way of evaluating the collision prediction ability of the digital twin?

1.4 Outline

Chapter 1: Introduction

This chapter provide an introduction to the thesis, the problem and the research questions.

Chapter 2: Background

This chapter serves to offer a comprehensive understanding of the technology itself and its various applications. It provides an overview of the digital twin concept and explores its practical uses. Additionally, the chapter delves into the simulator employed as the physical twin and explores its

functionalities. Furthermore, it discusses the creation of a dataset utilizing this simulator, highlighting its relevance and potential applications.

Chapter 3: Solution approach

This chapter presents the proposed solution and highlights the technologies utilized. It provides an overview of the key components and methodologies employed, setting the stage for the subsequent chapters.

Chapter 4: Implementation

This chapter is dedicated to the detailed implementation of the solution. It covers the integration of various algorithms, including machine learning and genetic algorithms, and outlines how they are employed in conjunction with the simulator.

Chapter 5: Results and discussion

This chapter presents the results of the conducted experiments that evaluated the proposed solution. The obtained findings are thoroughly analyzed and discussed in detail. This chapter provides insights into the performance and implications of the solution, shedding light on its effectiveness and limitations. At the end it answers the research questions.

Chapter 6: Conclusion

This chapter concludes the study by summarizing the findings and their implications. The limitations of the solution are acknowledged, and potential areas for future research are suggested. This chapter provides a concise overview and serves as the final reflection on the study.

Chapter 2

Background

In this chapter, we explore the practical applications of digital twins and provide an overview of their significance. The focus will be on the simulator employed in this study, including its development process and potential uses. Additionally, we delve into the dataset utilized for training and testing the digital twin solution, highlighting its role in ensuring reliable and accurate outcomes.

2.1 Digital twins

A digital twin serves as a digital representation of a physical asset. It acts as a counterpart to the physical twin, allowing communication between the two through data obtained from sensors and other sources. The digital twin utilizes this data to simulate possible scenarios and provide recommendations for optimizing the behavior of the physical twin. Moreover, the digital twin can also analyze the data to detect any anomalies or undesired occurrences that may require attention (Attaran and Celik, 2023).

What separates a digital twin to other similar models is its capabilities to communicate back and forth with its physical twin. A normal model is just a digital representations of something existing in the real world or a CAD model of something being created. There is also a possibility that a digital

model is getting information and data from a real object and processing that in some way, but that model is not communicating that back to the physical part. A digital twin will use continuously streamed data from the physical twin in a way that enhances and makes both twins coexist to solve a problem, detect one that might occur or speed up the production (Attaran and Celik, 2023, p. 2).

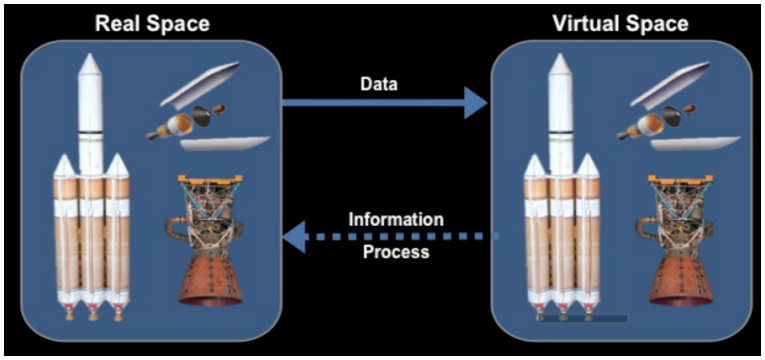

Figure 2.1: Communication between a physical and a digital twin.[1]

Digital twins can and are being used in many different scenarios and workplaces. That be it in agriculture, smart cities, manufacturing or infrastructure (Attaran and Celik, 2023, p. 4). The use cases are many and it can help with creating new and better solutions.

One of the first use cases of digital twins was NASA back in 1970 during the Apollo 13 mission (Allen, 2021). When an oxygen tank exploded and at the same time damaged the main engine, several simulations a lot of data transmissions were used to come up with a solution, which in turn were used to save the three people on board (Uri, 2020).

A digital twin can be used in multiple different scenarios, ranging from the development of an aeroplane, to small processes in a factory to an architect displaying a house to a customer virtually (Plank, 2019).

To illustrate the concept of digital twins within the context of a car man-

[1] Wilmjakob, August 30 2020, CC BY-SA 4.0
`https://commons.wikimedia.org/wiki/File:Digital_Twin_Concept_of_Grieves_and_Vickers.png`

ufacturer, Volvo's approach can be used as an example. Volvo has utilized the Unity game engine to facilitate collaboration and improve mutual understanding among different teams. By creating virtual environments and scenarios, designers and engineers can visually demonstrate their ideas and showcase how certain components or functionalities should work. This approach eliminates the limitations of verbal communication and allows for a more accurate and intuitive representation of concepts, enhancing overall communication and efficiency within the development process. (Unity, n.d.).

2.2 Self-driving cars

In recent years, self-driving cars have become a prominent topic of discussion, both online and offline. But what exactly is a self-driving car?

A self-driving or autonomous vehicle refers to a car that has the ability to operate and navigate without human intervention, similar to how a human driver controls a vehicle. It can autonomously control essential functions such as the throttle, brakes, and steering, essentially performing all the tasks that a human driver would typically handle. The key distinction is that a self-driving car operates without the need for direct human intervention.(Synopsis, n.d.)

Autonomous vehicles are classified into various levels, ranging from 0 to 5, each representing a different degree of autonomy. At level 0, the vehicle relies entirely on human control, while at level 5, the vehicle possesses full autonomy and can make independent decisions about its actions, regardless of the driving conditions. In the intermediate levels, human intervention is still possible, allowing individuals to take control of the vehicle if necessary, but not mandatory for all driving situations. (Synopsis, n.d.)

The specifics of these levels are as follows:

- Level 0: No Automation - The vehicle operates with full human control, and there is no automation present.
- Level 1: Driver Assistance - Basic automation features, like cruise

control, are available to assist the driver.

- Level 2: Partial Automation - The vehicle can simultaneously control certain functions, such as steering and acceleration, but the driver must remain engaged and ready to take over when needed.
- Level 3: Conditional Automation - The vehicle can handle most driving tasks under specific conditions, but the driver must be prepared to intervene when prompted by the system.
- Level 4: High Automation - The vehicle can operate autonomously in most situations, but it may still require human intervention in certain circumstances.
- Level 5: Full Automation - The vehicle is capable of complete autonomous operation in all conditions and environments, with no human intervention required.

Currently, many manufacturers, including Tesla with its Autopilot system, operate at Level 2, which involves partial automation. The significant difference lies in the levels beyond Level 2, where the vehicle may no longer require human intervention. These higher levels raise important discussions about certifications, rules, and ethics, but their detailed exploration is beyond the scope of this thesis.(Autocrypt, 2023)

2.3 SVL Simulator

In this thesis, the simulator used is the LGSVL Simulator, developed by LG Electronics America R&D Lab, formerly known as LG Silicon Valley Lab, situated in Santa Clara, California. The simulator has since been renamed as the SVL Simulator (Rong et al., 2020). The development of the simulator began in late 2018, and LG Electronics made the decision to end any further development at the beginning of 2022 (LG Electronics America R&D, 2022).

This simulator also has an open source library with a lot of information about how to use it. This also includes how to use different autonomous driving frameworks with the simulator (LG Electronics America R&D, 2021).

Figure 2.2: Before starting a simulation with the Python API with the original SVL Simulator.

The simulator was developed because there was a need for a platform where autonomous vehicles or robots could be developed. Here, the simulation can recreate real-life scenarios with high fidelity (Rong et al., 2020, p. 1). This includes how the multiple sensor that are mounted on a vehicle perceives what is around it to how traffic flows with multiple different vehicles like cars and trucks to also pedestrians walking around. Another benefit of developing a simulator for use with an autonomous driving system is the increased safety it provides compared to the real-world counterpart, where accidents can have severe consequences. The simulator also has multiple maps to use where many of them are digital twins of a real world location (Rong et al., 2020, p. 6). When running the simulator, various parameters can be adjusted to replicate desired weather conditions, such as time of day, rain, fog, and road wear. Additionally, the SVL Simulator distinguishes itself by allowing the creation of various vehicles with diverse sensor configurations.

The main vehicle in the simulator is called the ego vehicle, which can be controlled by either an autonomous framework or the user. It is equipped with various sensors and can be customized with different configurations. Other moving objects, such as cars, trucks, buses, and pedestrians, are referred to as NPCs (non-playable characters). NPCs populate the surrounding area of the ego vehicle, adding complexity to the environment.

2.3.1 WISE

To utilize the simulator, SVL developed a web user interface called (WISE). WISE enables users to create and launch various scenarios on a cluster where the simulator is hosted. Leveraging cloud technology, WISE facilitates the downloading of assets such as maps, vehicles, and sensors to the simulator. Additionally, the interface provides the capability to view simulation results and access relevant information pertaining to the simulations.

The WISE interface is designed to offer extensive customization options for various settings, including weather conditions, traffic patterns, density, and road networks. When it comes to vehicles, maps, and sensors, users can either create them from scratch or customize existing ones to suit their specific preferences. This versatility makes WISE an excellent tool for maximizing the potential of the simulator, allowing users to tailor the simulation experience according to their needs and objectives.

Within this interface, users also have the option to select different templates for the simulations they wish to run. These templates include random traffic, visual scenario editor, Python API, or API only. In the context of this thesis, the Python API template is utilized to manipulate the simulator with a wide range of available options. When launching a template in WISE, the corresponding settings are loaded into the simulator and the simulation begins. In cases where internet access or the web interface is unavailable, users can still choose and launch previously loaded scenarios/templates from within the simulator, provided they were previously loaded and run with WISE.

WISE also includes real-time visualization tools to monitor the simulation's performance as it happens and has the possibility to generate reports at the end of the simulation.

As mentioned earlier, the development of the simulator was concluded in 2022. However, the team made efforts to ensure that WISE remained operational for starting simulations. Initially, they had planned to maintain it until at least mid-2022, but in practice, they continued to keep it up and running for several weeks into 2023.

2.3.2 OSSDC-Sim

In the same message that the SVL team mentioned the discontinuation of the WISE website, it was mentioned that the simulator source code could maybe be forked to another project where the dependency to WISE was not needed.

This is exactly what was done by the people behind the Open Source Self Driving Car Initiative. They called the "new" simulator for OSSDC-Sim: An Autonomous Vehicle Simulator (forked from LGSVL Simulator). This is almost the same as the original SVL Simulator, but this time without the need for WISE. It is also worth mentioning that also this is an open source library which can be found on GitHub (Open Source Self Driving Car Initiative, 2022).

Figure 2.3: Before starting a simulation with the Python API with the original SVL Simulator.

When starting the simulator, the only option now is to use the Python API. When running the Python program, the simulator will automatically download any required assets if they haven't been downloaded before. After that, the simulation will start with the predetermined settings.

Figure 2.4: Launching the simulator.

In other words, this is the simulator that is used in the thesis. If WISE was not shut down, the OSSDC-Sim would not have been needed.

2.3.3 Python API

One of the primary motivations for utilizing this simulator in the thesis is the availability of the Python API. The SVL Simulator's Python API simplifies the process of initiating a simulation by allowing the settings and options to be defined using Python. The simulator's development team has provided a comprehensive documentation page that guides users on utilizing the API and modifying settings and assets according to the programmers' specific requirements and preferences. (LG Electronics America R&D, 2020).

The API also incorporates numerous scenarios that can be launched individually, allowing users to experiment with the simulator and explore its capabilities. These predefined scenarios serve as valuable resources for users to gain hands-on experience and discover the range of possibilities offered by the simulator.

The API establishes communication with the simulator (OSSDC-Sim) using websockets, enabling the initiation of simulations and the exchange of messages. The "run()" function serves as a critical component within a Python program that interfaces with the simulator. This function determines the duration of the simulation in seconds and controls the sampling rate of the simulator.

The simulator then sends messages back to the Python program through "Callbacks" and in between calls of the "run()" function.

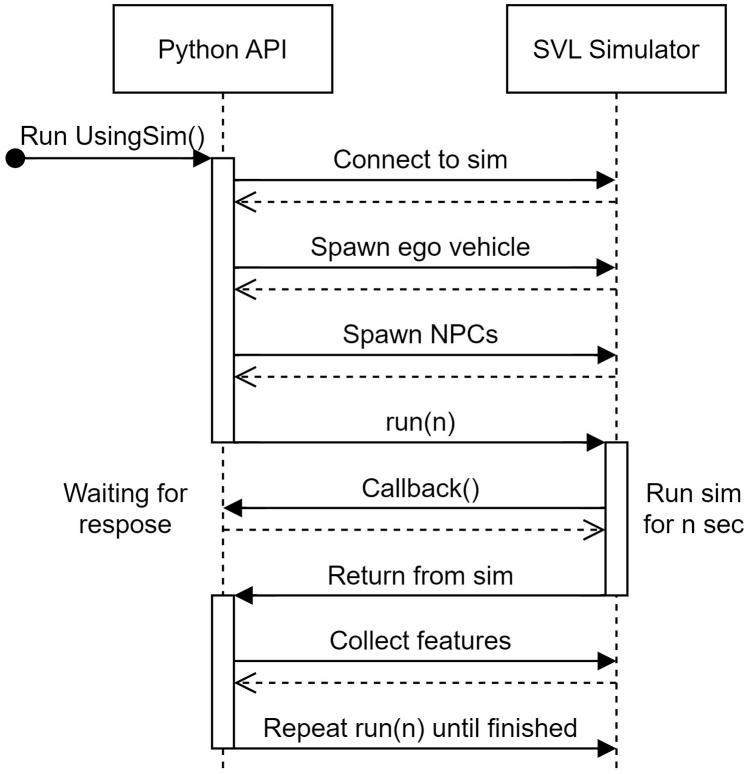

Figure 2.5: An overview over how the simulator and the API communicates.

Ego vehicle

The primary vehicle in the simulator is the ego vehicle, as previously mentioned. With the Python API, it becomes possible to control the vehicle through various functions, either manually or by utilizing an autonomous driving system. In terms of manual driving, it is feasible to automate a significant portion of it by leveraging the vehicle's multiple sensors and utilizing the additional functions offered by the API. With the assistance of the API, the vehicle can be transformed into a self-driving entity.

The vehicle is equipped with various sensors, with the main ones including a camera, LiDAR, IMU, GPS, Radar, CanBus and VideoRecording. It is also possible to create custom sensors, but this is not done in this thesis. From the mentioned sensors, only the LiDAR sensor is used to a large degree.

About the LiDAR sensor

The LiDAR (Light Detection and Ranging) sensor is an essential component that utilizes light to determine the distance to objects. In basic terms, it emits a brief laser pulse, and if the light encounters an obstacle, it will bounce back to the emitter. By measuring the time it takes for the light to return, and knowing the constant speed of light, the sensor can calculate the distance to the object, as depicted in 2.6 below. Although the LiDAR sensor has additional applications beyond distance measurement, they are not directly relevant to this thesis.

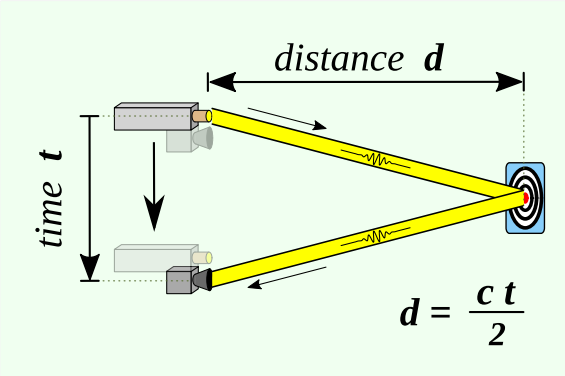

Figure 2.6: Explanation about how a LiDAR calculates the distance to an obstacle.[2]

When figuring out the distances to everything around the sensor, it will generate a three-dimensional image of its surroundings. It does this by rotating fast around its own axis and using multiple lasers which are aligned vertically with different angles. This results in a vertical field of view that are rotated 360 degrees to create a the three dimensional image. How many lasers in this vertical field of view and how many times each laser emits a pulse per rotation will determine the resolution of the image. The resulting file from this is also called a point cloud.

[2]RCraig09, May 1 2020, CC BY-SA 4.0
`https://commons.wikimedia.org/wiki/File:20200501_Time_of_flight.svg`

Figure 2.7: A normal view from within the simulator. Here one can see the ego vehicle and an NPC vehicle.

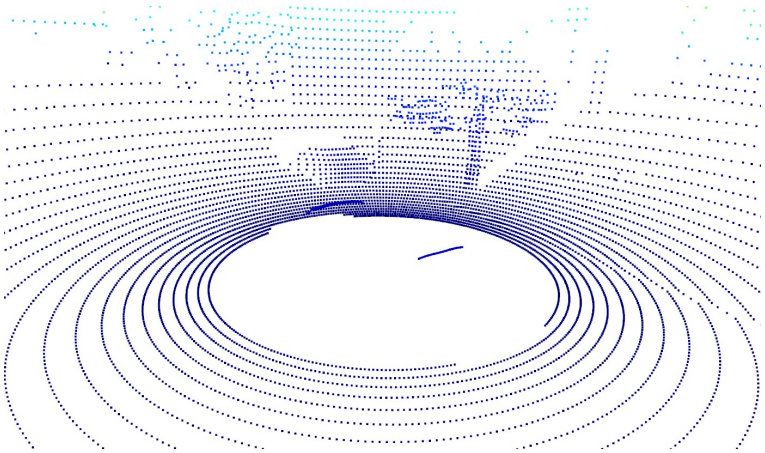

Figure 2.8: How the LiDAR sees the same image as in figure 2.7. Notice how the sensor captures the back of the car and the closest tree.

Non playable characters

As mentioned, there is also NPCs that can be deployed and manipulated in the simulator. There are multiple different available vehicle types and a

multitude of variations of a pedestrian. NPCs can be randomly spawned at various locations within the environment, moving autonomously on their own. However, they can also be extensively controlled, including their spawn locations and intended destinations. Additionally, waypoints can be created for NPCs to follow, specifying the desired paths and speeds at which they should move towards specific points.

2.4 Deep Scenario

In this thesis, some data that are used is collected from the Deep Scenario dataset (Lu et al., 2023). The data is created with different strategies and driving conditions. This results in over 30,000 different scenarios where about 1,000 of them is ending with a collision with either a car, pedestrian or a different object. All the different scenarios are executed on the SVL Simulator and are designed in a way that allows for their recreation later using the Deep Scenario toolset.

2.4.1 Scenarios

The different scenarios are first split into three strategy branches, rl-based, greedy and random, then further into the reward branches dto, ttc and jerk. There are also four different roads and four different weathers. The weathers are both night and day where they each have one occurrence of rain and sun, or at least a clear sky. Each scenario also has each own deepscenario-file, which is a file that is generated from the simulator. This file has information about some of the setting on the simulation and the position and speed of the multiple objects, both the ego vehicle and the NPC's that might be in the scenario. The speed and position is noted for six timestamps for each object. The time between each timestamp is noted at the top of each file and is 0.5 for all of them. This then means that the time between each timestamp in the scenario is 0.5 seconds. Each scenario is also represented as a row in a csv-file. The row consist of some attributes that sum up how the scenario played out.

Each scenario has been driven with an autonomous driving system (ADS),

more specifically with Apollo 5.0 and on version 2021.1 of the SVL Simulator.

2.4.2 Features

Each scenario in the dataset has multiple features. The first three are "Execution", "Scenario_ID" and "Configuration_API_Description". The first one is the execution ID. The second one is the identity of each scenario (which can be used to collect the deepscenario-file 2.4.3), which also tells which scenario it is and its execution number. The third one is a short description of what happens. The final six features, are features that are collected from the scenario itself. "Attribute[TTC]", time to collision, is a safety measure that says something about how long time it is to a potential collision. "Attribute[DTO]", distance to obstacles, tells how much distance there is to an obstacle. "Attribute[JERK]", change in acceleration, is a measure that tells something about the passengers' comfort. "Attribute[COL], collision, is either true or false and is true if a collision has happened during the scenario. "Attribute[COLT], collision type, describes which type of object the ego vehicle collided with if that happened, that be it a car, pedestrian or something else. "Attribute[SAC]", speed at collision, tells the speed the ego vehicle had when and if a collision happened. These features can be extracted from the csv-files and a peek at this data can be seen in table 2.1.

	Execution	ScenarioID	Configuration_API_Description
0	16	16_scenario_6	A skyblue Hatchback is driving ahead (far) of ...
1	16	16_scenario_7	A pedestrian is crossing the road from left to...
2	16	16_scenario_8_pedestrian	A skyblue Hatchback is driving ahead (far) of ...
3	17	17_scenario_0	A black SchoolBus is crossing the road (far) a...
4	17	17_scenario_1_npc_vehicle	A black SchoolBus is overtaking (near) the ego...

	Attribute[TTC]	Attribute[DTO]	Attribute[Jerk]	Attribute[COL]	Attribute[COLT]	Attribute[SAC]
0	1.955202	22.964591	9.98	False	None	0.000000
1	1.924612	6.350632	8.20	False	None	0.000000
2	13.859044	1.473800	6.02	True	pedestrian	5.225631
3	0.925033	11.506568	5.36	False	None	0.000000
4	0.467288	3.853776	5.40	True	npc_vehicle	0.004282

Table 2.1: A peek at the data stored in the csv-files.

2.4.3 Deep scenario XML-files

Each scenario also has its own .deepscenario file. This file is in an XML, but with the ".deepscenario" extension instead of the normal ".XML". The file consists of multiple elements, where the main ones are "environment", "entities" and "storyboard".

Environment is the where and when of the scenario and entities says which object that are spawned in, that being the ego vehicle and up to several NPCs. In the storyboard element is where the rest of the information is; where everything spawns and how they are moving around inside the simulator. This information is given by the position in an x-, y-, and z-coordinate and with a GPS position along with the rotation of the objects. The velocity is noted in three different ways, that is the velocity in three dimensions, the combined velocity as speed and the angular velocity. Each of the just mentioned features are noted in six different "waypoint" elements where they have a "timeStamp" value from one to six. All the objects that are spawned in the scenario have these elements. An over

```
1  <DeepScenario timestep="0.5">
2      <FileHeader author="Greedy_Strategy" ...
           date="2021-11-27T10:00:00" ...
           description="DeepScenario Format" ...
           simulatorVersion="2021.01"/>
3      <Environment>
4          ...
5      </Environment>
6      <Entities>
7          /// Spawn objects
8      </Entities>
9      <StoryBoard>
10         <Initialization>
11             <ObjectInitialization objectRef="Ego0">
12                 /// Position information
13             </ObjectInitialization>
14             <ObjectInitialization objectRef="NPC0">
15                 ///...
16             </ObjectInitialization>
17         </Initialization>
18         <Story name="Default">
19             <ObjectAction name="Act_Ego0">
20                 <objectRef objectRef="Ego0"/>
21                 <WayPoint timeStamp="1">
```

```
22                      <DynamicParameters>
23                          ...
24                      </DynamicParameters>
25                      <GeographicParameters>
26                          ...
27                      </GeographicParameters>
28                  </WayPoint>
29                  ...
30                  <WayPoint timeStamp="6">
31                      ...
32                  </WayPoint>
33              </ObjectAction>
34              <ObjectAction name="Act_NPC0">
35              </ObjectAction>
36          </Story>
37      </StoryBoard>
38  </DeepScenario>
```

Chapter 3

Solution Approach

In this chapter it will be described what the different parts to answer the research questions are. The chapter goes into more depth about what the parts are, what can or has been done and why they are implemented.

3.1 Generated collision data

During the work on the thesis, it became clear that the data from the Deep Scenario dataset was not detailed enough to train a model that can be used for collision detection. This can for instance be seen by looking at the correlation between Deep Scenario and the generated data in table 3.1.

DeepScenario		Self-generated	
Feature	Correlation	Feature	Correlarion
Attribute[TTC]	-0.1069	TTC1	-0.1012
Attribute[DTO]	-0.01162	TTC2	-0.1771
Attribute[Jerk]	0.10521	TTC3	-0.2515
speed1	0.11594	TTC4	-0.3163
speed2	0.0979	DTO1	-0.0626
speed3	0.0371	DTO2	-0.1098
speed4	-0.0381	DTO3	-0.1573
speed5	-0.0985	DTO4	-0.2086
speed6	-0.1321	Speed1	0.1546
		Speed2	0.1709
		Speed3	0.1740
		Speed4	0.1688

Table 3.1: The correlations for the multiple features and from both datasets regarding collision. The feature type with an average below 0.1 is removed from the table.

The data consists of the mentioned features in section 2.4.2. The difference between the two datasets are the amount features back in time for each feature. In Deep Scenario, the features are TTC, DTO, JERK and speed and angular velocity six time steps back in time. Since it is difficult to pin point exactly when the collision happened, it is just one collision value per scenario.

In the generated collision data, all the features, TTC, DTO, JERK, speed, angular velocity and collision have values n time steps back in time. Since this data was generated during the writing of the thesis, it is possible to register exactly when a collision happened, and because of this, it is possible to note that the m time steps (or rows) which lead to a collision can be noted as a collision as well. By creating the data in this way there are many more collisions in regard to not a collision when comparing it with Deep Scenario. This leads to less unbalanced data, which will we touched upon in section 4.4.1.

During the creation of the data, the ego vehicle were driven around in the simulator and storing all the needed data. The car was driven for a total of approximately 35 minutes, and had a total of 215 individual collisions.

	Time	TTC	DTO	JERK	Speed	asX	asY	asZ	COL
2401	12.0	13.438	20.4665	0.196	16.427	0.000	0.00	-0.000	0
2402	12.5	15.222	19.8645	0.008	16.226	0.000	0.00	-0.000	0
2403	13.0	15.046	19.2738	0.008	16.027	-0.000	0.00	-0.000	0
2404	13.5	19.378	18.8359	0.028	15.835	0.000	0.00	0.000	0
2405	14.0	1.174	13.2444	0.008	15.645	-0.000	0.00	0.000	0
2406	14.5	0.350	5.4750	0.016	15.451	-0.000	0.01	0.000	0
2407	15.0	0.053	0.8400	52.528	2.125	0.116	-0.10	0.065	1
2408	15.5	50.000	2.0090	52.320	1.879	-0.040	-0.00	-0.019	1
2409	16.0	50.000	3.3947	0.920	1.863	-0.000	-0.00	-0.000	0

Table 3.2: Raw self-generated data. The processed data can be seen in table 4.1.

After the creation of the data, it had to be made usable for training the models. This is explained in more detail in section 4.2.2.

3.2 Machine learning for collision prediction

A major part of our research focuses on collision prediction within the digital twin framework, and this section focuses on the foundations of machine learning. By delving into these foundational concepts, a comprehensive understanding of the underlying techniques and methodologies is established, allowing for a greater understanding in the decisions behind the collision predicting digital twin.

3.2.1 Introduction to machine learning

Machine learning plays a crucial role in collision prediction within the digital twin framework. This section provides an overview of machine learning, highlighting its importance in collision prediction. Machine learning, an influential field of artificial intelligence, enables systems to learn and make predictions without explicit programming. In the context of collision prediction, it facilitates the automatic identification of patterns and extraction

of insights from complex datasets. Consequently, it enables accurate predictions of collision events in autonomous driving scenarios.

In the context of collision prediction, various machine learning approaches are applicable. This research specifically emphasizes the utilization of supervised learning as our primary focus for achieving accurate predictions. Supervised learning involves training models using labeled examples. These examples consist of input data that represents driving scenario features, accompanied by corresponding labels indicating collision occurrences. By analyzing patterns and relationships within the labeled training data, supervised learning models can provide precise real-time predictions of potential collisions.

In this research, the performance of various machine learning models for collision prediction is investigated. Although these models were not explicitly designed for collision prediction, they have been extensively utilized and proven effective in diverse machine learning applications. By evaluating these models and training them on a comprehensive dataset of labeled collision scenarios, the aim is to assess their suitability and enhance collision prediction within the digital twin framework.

3.2.2 Data preprocessing and feature engineering

Data preprocessing is a crucial step in preparing the dataset for collision prediction analysis and modeling. It involves transforming raw data into a suitable format that is compatible with machine learning algorithms and enhances data quality.

By addressing noise, missing values, and inconsistencies, data preprocessing improves the reliability and consistency of the dataset. Techniques such as identifying and handling missing values, correcting inconsistencies, and reducing noise contribute to a more reliable dataset. Additionally, data preprocessing ensures compatibility with machine learning algorithms by scaling numerical features, encoding categorical variables, and transforming the data into a suitable format. These techniques enable efficient processing and seamless integration with the chosen algorithms.

One of these techniques is Label Encoding, which transforms categorical

variables into numerical values. This is necessary because machine learning algorithms typically work with numerical data. By assigning a unique numerical label to each category within a categorical variable, Label Encoding enables the algorithms to process and analyze these features effectively.

To further enhance the preprocessing stage, Standard Scaling, also known as feature scaling or normalization, is a technique commonly used to bring numerical features onto a consistent scale. It rescales the values of the features to a predefined range, such as between 0 and 1 or with a mean of 0 and standard deviation of 1. This normalization process prevents certain features from dominating the model's learning process due to their larger magnitudes. By ensuring all features are on a similar scale, Standard Scaling enables fair comparisons and promotes balanced learning within the machine learning algorithms.

Another technique that was employed is undersampling. This technique involves reducing the samples of the majority class to balance the class distribution, particularly useful in situations where the dataset is imbalanced. By randomly selecting a subset of instances from the majority class, undersampling helps prevent the model from being biased towards the majority class and encourages better learning of the minority class. It addresses the issue of imbalanced data and improves the performance and reliability of the machine learning models.

Overall, data preprocessing plays a vital role in optimizing the dataset for collision prediction. By improving data quality and ensuring compatibility with machine learning algorithms, it enhances the dataset's quality, reliability, and compatibility for accurate and effective collision prediction analysis and modeling.

3.2.3 Machine learning models for collision prediction

Machine learning models are important for predicting collisions in the digital twin framework. They analyze large amounts of data from sensors and sources to make accurate predictions about potential collisions. These models are great at finding patterns and relationships that traditional methods might miss. This is crucial for understanding and reducing collision risks, especially in complex driving situations. By handling complex and non-

linear relationships, machine learning models provide valuable information about collision risks, making autonomous driving systems safer and more reliable.

In the following sections, an exploration of the machine learning models employed for collision prediction within the digital twin framework will be undertaken. We will explore the application of MLPClassifier, Random Forest Classifier, SVM Classifier, and XGBoost Classifier, as these models offer unique strengths and capabilities for accurate collision prediction.

Multilayer Perceptron

The exploration of machine learning models for collision prediction within the digital twin framework began with the Multilayer Perceptron (MLP) Classifier, a type of artificial neural network. This model, with its multiple layers of interconnected nodes and the application of a nonlinear activation function, excels at identifying patterns within large, high-dimensional datasets. This makes it well-suited to the collision prediction task, where a multitude of factors such as vehicle specifics like speed, jerk, and angular velocity, proximity to other objects, and environmental conditions, including weather and lighting changes may interplay in complex ways. MLP's ability to model such intricate nonlinear relationships enhances its value in this study (h20.ai, n.d.).

Random Forest

Following the MLP Classifier, we employed the Random Forest Classifier. This model, an ensemble learning method, operates by integrating multiple decision trees, thus offering a more comprehensive and robust output. The Random Forest Classifier's ability to process large datasets, its resistance to overfitting, and its capacity to handle both numerical and categorical data make it particularly useful for the collision prediction task (Javatpoint, n.d.).

Support Vector Machine

Continuing the exploration, the Support Vector Machine (SVM) Classifier was incorporated into the study. The SVM Classifier is a powerful algorithm known for its ability to handle complex datasets and effectively capture nonlinear relationships. It works by mapping the input data into a high-dimensional feature space and finding an optimal hyperplane that separates different classes, maximizing the margin between them. By finding the most discriminative hyperplane, the SVM Classifier can accurately predict collision events based on input features such as vehicle attributes, environmental conditions, and proximity to other objects. The robustness of the SVM Classifier, coupled with its ability to handle both numerical and categorical data, contributes to its effectiveness in the collision prediction study (Scikit-learn, n.d.).

XGBoost

Lastly, the study incorporated the XGBoost Classifier. This machine learning framework, based on the gradient boosting decision tree algorithm, has demonstrated remarkable efficiency and effectiveness in handling large datasets and capturing complex data interactions. The XGBoost Classifier leverages the power of gradient boosting to iteratively add decision trees to the model, with each subsequent tree learning from the mistakes made by the previous trees. This sequential approach enhances the model's predictive capabilities and enables it to effectively capture intricate patterns and relationships within the collision prediction data. The XGBoost Classifier brings a unique strength to our modeling approach, enhancing our strategy and contributing to more accurate collision predictions. (Seif, 2019)

In summary, the machine learning models utilized in our collision prediction study within the digital twin framework have shown promising results. However, evaluating their performance using traditional metrics has posed certain challenges, which we will delve into in the next section.

3.2.4 Evaluation and performance metrics

This section focuses on key metrics that assess the performance of the models used in this study. These metrics, including the confusion matrix, accuracy, precision, recall, and the F1 score, serve as critical tools to evaluate the effectiveness of the models. Each metric provides a different perspective on the model's predictive abilities. By comprehending these metrics, a deeper understanding of the model's performance can be obtained. This section will also shed light on a notable discrepancy observed between the results of the evaluation metrics and the actual performance of the models. A deeper dive into this interesting discrepancy is set to be the focus of the following section.

A confusion matrix, often used in predictive analytics, is a table with two rows and two columns that reports the number of true positives, false negatives, false positives, and true negatives. This layout gives the matrix the power to provide a more nuanced evaluation of a model's performance than simply noting the proportion of correct classifications, which is known as accuracy. The importance of a confusion matrix becomes particularly apparent when dealing with unbalanced datasets, where the distribution of classes is uneven. In these cases, accuracy alone could be misleading, making a confusion matrix a more reliable tool. The confusion matrix accommodates four possible outcomes for every prediction made by the model. A true positive (TP) is when the model correctly identifies a positive sample, while a true negative (TN) is when a negative sample is accurately recognized. On the other hand, a false positive (FP) is when a negative sample is incorrectly identified as positive, and a false negative (FN) is when a positive sample is misclassified as negative. Each of these outcomes contributes to a comprehensive understanding of the model's performance, offering valuable insights that can guide further refinement of the model (Brownlee, 2016).

There are other metrics that can be derived from a confusion matrix, each of which carries its own significance and utility.

One of the crucial metrics used in binary classification tasks is accuracy. Specifically, it is the total number of correct predictions (both true positives and true negatives) divided by the total number of predictions made. In other words, accuracy gives us a straightforward idea of how often the model

is correct in its predictions. The formula for calculating accuracy is:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.1}$$

True positives and true negatives are the cases where the model's predictions match the actual classes, while false positives and false negatives are the instances where the model's predictions don't align with the actual classes (Ping Shung, 2018).

The accuracy metric provides a general understanding of the model's reliability in predicting collisions. High accuracy suggests that the model's predictions align well with actual outcomes. However, it's important to keep in mind that accuracy alone does not tell the whole story. It gives an aggregate view of the model's performance, but it does not provide insight into how well the model performs on individual classes - collisions and non-collisions. To gain a more nuanced understanding of the model's performance, we turn to precision.

Precision, also referred to as positive predictive value, is a critical metric in the domain of machine learning.. It quantifies the ratio of relevant instances among those instances identified by the model. In essence, precision measures how many of the instances that the model classified as positive were actually positive. It is calculated as the number of true positives divided by the sum of true positives and false positives, as seen in equation 3.2. For example, if a model predicts that ten instances are positive and eight of these predictions are correct, the precision would be 0.8, or 80% (Ping Shung, 2018).

$$precision = \frac{TP}{TP + FP} \tag{3.2}$$

Precision gauges the model's ability to avoid false alarms. In other words, it measures how many of the predicted collisions were indeed collisions. In a traffic context, a high precision model is valuable as it minimizes unnecessary preventive measures that might otherwise disrupt the smooth flow of vehicles. Yet, while precision concerns the model's credibility when it

predicts a collision, it does not account for collisions that the model fails to predict. That's where recall comes into play.

Recall, in the context of machine learning, measures the proportion of actual positive instances that were correctly identified by the model. It is defined as the number of true positives divided by the sum of true positives and false negatives, and has the following formula:

$$recall = \frac{TP}{TP + FN} \tag{3.3}$$

For instance, if there are ten actual positive instances and the model correctly identifies seven of them, the recall would be 0.7, or 70% (Ping Shung, 2018).

In the context of this thesis, recall assesses the model's effectiveness in capturing all potential collisions. High recall is vital to ensure safety as any missed collision could lead to severe consequences. While both precision and recall provide valuable insights, they focus on different aspects of the model's performance. Hence, a measure that combines these metrics is often desirable. The F1 score fulfills this need. It provides a balanced view of precision and recall, becoming particularly useful in scenarios like collision prediction where both false positives and false negatives have significant implications. The formula for F1 is seen in equation 3.4.

$$F1 = 2 \cdot \frac{precision + recall}{precision \cdot recall} \tag{3.4}$$

Another important tool used in this thesis to evaluate the performance of the collision prediction models is ROC - Receiver Operating Characteristic. The ROC curve plots the true positive rate (sensitivity) against the false positive rate (1-specificity) at various classification thresholds. This curve provides a visual representation of the trade-off between true positive rate and false positive rate, allowing for the selection of an optimal threshold that balances the model's sensitivity and specificity. (Brownlee, 2018)

In the context of this study, the ROC curve helps assess the model's ability to distinguish between collision and non-collision instances. Furthermore,

the area under the ROC curve (AUC) is utilized as a summary metric to quantify the overall performance of the model. A higher AUC value would suggest a better ability of the model to correctly classify collision and non-collision instances.

While these statistical metrics offer valuable insight into the model's performance, they do not necessarily capture all aspects of real-world functionality. Practical considerations, such as the timing of the collision prediction and the model's subsequent reaction, play a crucial role in determining the model's overall effectiveness. It's observed that a model with the highest score in a particular metric does not always translate into the best performing model in a real-world scenario. For example, a model with a lower recall might outperform in practice compared to a model with a higher recall. To complement these metrics, testing the model in a real-world simulator is of utmost importance.

This additional evaluation step allows for a more comprehensive assessment of the model's performance, providing insights that closely resemble those it will encounter in actual scenarios. This striking observation about the potential discrepancies between statistical metrics and simulator performance raises interesting questions that warrant further investigation in subsequent section.

3.3 Genetic algorithm

In order to be able to generate more data for the model, a genetic algorithm has been created and used. A genetic algorithm can be efficient in many different fields and has en excellent ability to search and figure out the best set of variables to solve a problem. Especially when going through every possible solution is unfeasible, one such algorithm can find a close to optimal solution. (Harman and Jones, 2001)

The algorithm itself is inspired by Charles Darwin's evolutionary theory, where the main point is that the most fit individuals will be the ones that reproduce (Sivanandam and Deepa, 2008, p 15). This point also applies to a genetic algorithm. Since the most fit individuals reproduce, this also implies means that there is something that determines the fitness of the population,

where the most fit individuals have the highest chance of passing their genes down to the next generation.

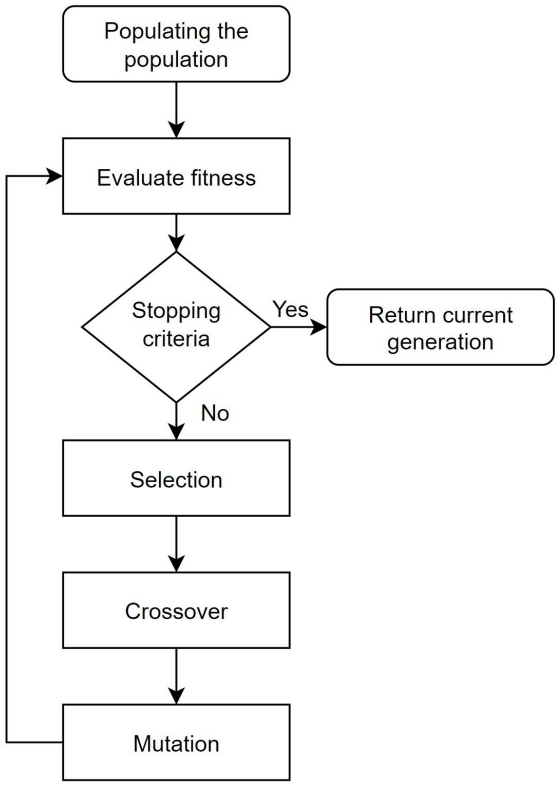

Figure 3.1: How the different parts of a genetic algorithm follow each other. In this case the stopping criteria is either that the population is fit enough or that the generation limit is reached.

3.3.1 Simple implementation example

There are many different versions of the algorithm, but the primary parts are the starting population, fitness evaluation, selection, crossover and mutation.

At the beginning, the individuals in the population are created. Each in-

dividual consists of a set of variables, or genes as they are known within this algorithm. At the start each gene is set to be a random value between an upper and a lower limit. After that, each individual is sent through the fitness functions and gets a score, then the most fit individuals gets to reproduce and the least fit individuals ceases to exist in the selection part. To be able to produce the next generation, there is a function known as the crossover function that determines which genes from the parents that are being sent down to the child. The next step is the mutations part, where there is a chance that one or more of the genes of the child is mutated and gets a new value. Because of the mutation, new genes gets introduced to the gene pool and the population keeps a certain diversity. Together this might be helping the population's top individuals to reach an even higher fitness score or that some individuals with a lower score to reproduce so that the next generations gets better. (Kour et al., 2015)

Eventually, the algorithm will have created the maximum number of generations or a certain fitness score has been reached. The top individuals can then be retrieved and used.

3.3.2 Variations of the algorithm

What is mentioned above is a very basic version of a genetic algorithm. There are many different variations of each part of the algorithm.

Population

The initialization of the population can happen in multiple ways. One of them is as mentioned in the simple implementation 3.3.1, that all the genes in each individual is randomized. This grants a large diversity in the gene pool, but the downside is that the algorithm can use a long time to find the best solution to a given problem. (Syberfeldt and Persson, 2009)

A different method of populating the population can be in a heuristic manner. That means that the genes in for each individual can be something that one can believe might be close to the solution and in that way try to make the algorithm converge in less time. The downside of this can be that

a lot of the individuals are very close to each other regarding the genes, which again says something about the diversity in the gene pool. Another thing that might happen is that the algorithm might have a premature convergence, which means that it returns a solution before the best solution is found. The next generations is also not able to produce offspring that is superior to the one before it. This scenario is also known as a local minima. (Shyalika, 2019)

By using a heuristic approach at the start of the algorithm one might also get good and fast results, especially if the genetic algorithm uses few generations. Something to know when using this method of initialization is that it might be difficult to find those good starting values REF smash. If it is planned to have a large number of generations, the results of a good heuristic starting population will have less effect than if the there are few generations. (Syberfeldt and Persson, 2009)

Fitness evaluation

To be able to find out if an individual is good or not, a fitness function is needed to figure that out. This function is very specific to each problem a genetic algorithm is used for. There are also several things one should think of when creating a fitness function. A fitness function should be easy to understand and to get the gist of why it evaluates what it is. Since the function is used a lot, one time per individual per generation, it should be fast to calculate and not use too many resources. Therefore, how fast the genetic algorithm is, is influenced quite a bit by the time the function uses to calculate. The results from the function should also be easy to understand, for example that a low number should equal a bad score and that a high number should equal a good score or vice versa. (Mallawaarachchi, 2017)

Selection

There are multiple ways of determining which parents that should be able to reproduce for the next generation. What often is the point with this function is to give the individuals with the highest fitness score the highest probability to reproduce. Another thing to think about is that the most fit

individual might be in a local minima, so it is important to give the other individuals a chance at pass on their genes as well (Shukla et al., 2015). It is also possible to create offspring with more than two parents.

One of the ways to select the parents, is with a use of a roulette wheel. There the individual with the highest fitness score will have the largest section of the wheel, the second fittest will have the second largest section and so on. By implementing the selection in this way, all individuals will have a chance to become a parent, but the individuals are rewarded by having a good fitness score. The probability of being chosen is then the individuals fitness score divided by the total sum of all fitness scores in the current generation (Shukla et al., 2015) and can be seen in equation 3.5 below.

$$P_sel(a_i) = \frac{f(a_i)}{\sum_{i=1}^{n} f(a_j)};\ j = 1, 2, \ldots, n \tag{3.5}$$

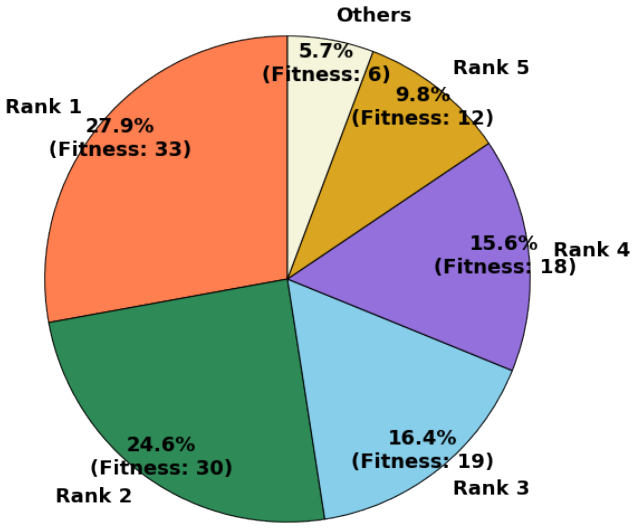

Figure 3.2: Showing the distribution of the individuals in a roulette wheel.

Another way of implementing the function is to use rank of the individuals

in stead if the fitness score directly. In this way all the individuals gets a score and they are then sorted based on that where the most fit individual gets the best rank. The rank is then used to select a parent either in a linear or a exponential way. If linearity is chosen, the ranks are used so that the chance of picking a parent gets linearly worse the longer down the ranks you get. In contrary to the exponential way where the chances of getting picked gets exponentially lower the worse rank the individual has. (Shukla et al., 2015)

A third method of implementation is by picking a parent with a tournament selection, where the winner is the parent. This is one of the most popular methods to use, as it is both efficient and easy to implement. To begin, a predefined number of individuals gets picked at random from the entire population. Then the most fit of those individuals is picked as the parent Shukla et al., 2015. It is often used just two participants in a tournament, this will also give a linear distribution of the parent, as seen in figure 3.3 together with some other amount of participants.

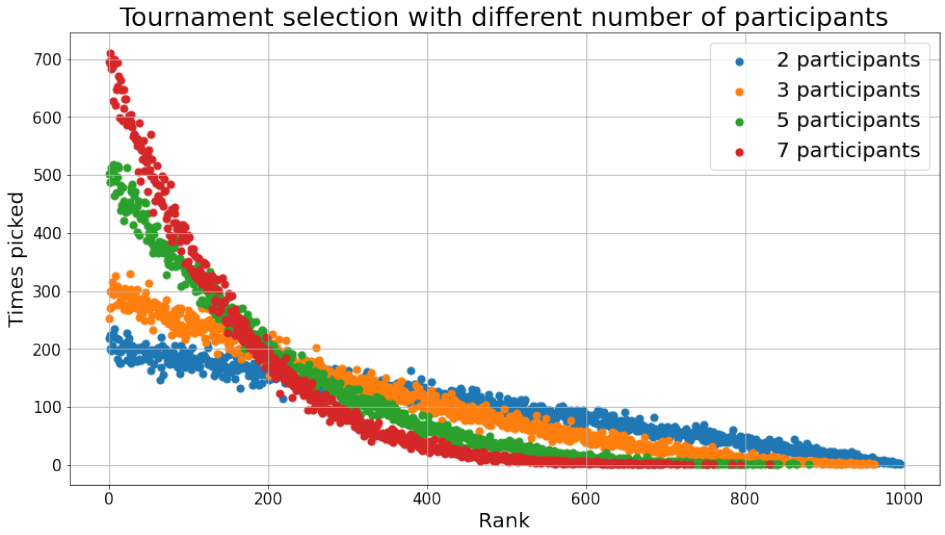

Figure 3.3: Distribution of which rank is picked when different number of participants are used.

By choosing good parents with a good selection method, the algorithm gets the best use of its population. A good method will also keep the gene pool

diverse and thus keep a lot of potential good genes still a chance to combine and become fit individuals.

Crossover

Crossover is as mentioned a method that is suppose to mix the genes from the parents to produce offspring. A good crossover function should do this in a way that the produced children can have a chance to be better than its parents, even though the child can get an unfortunate combination from the parents.

There are also here multiple ways to do this. The common point crossover is one of them. There both parent's chromosome gets a slice in the middle, where child A receives part one from parent A and part two from parent B. Child B will then get part 1 from parent B and part two from parent A. It is also possible to use multiple of these splits to shuffle the genes a bit more. (Umbarkar and Sheth, 2015, p. 1083)

Another crossover function is the uniform crossover. In this function each gene is chosen by a uniform random number, either 1 or 0. If the number is 1, child A gets the gene from parent A and child B gets the gene from parent B or vice versa. This then repeats until all the genes are distributed. (Umbarkar and Sheth, 2015, p. 1084)

There is also an option to choose a function that takes the average value of each gene from two parents. This function produces only one child from two parents, so it is needed to find twice as many parents to reach the population limit in contrary to the crossover functions mentioned above. (Umbarkar and Sheth, 2015, p. 1084)

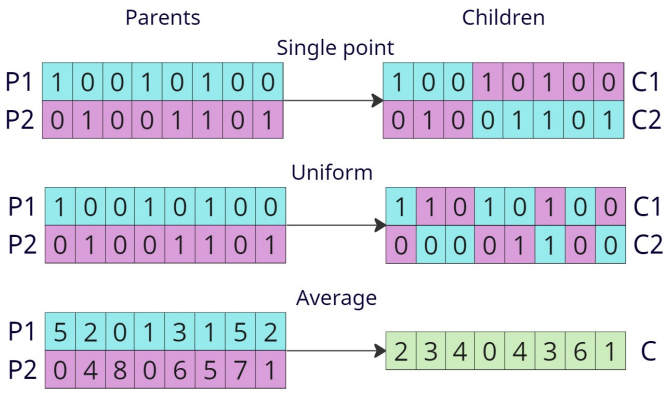

Figure 3.4: Three different crossover functions, single point, uniform and average.

Mutation

The mutation function has the responsibility to bring new genes into the gene pool. This contributes to keep the diversity and also creates new genes that might make the population more fit. Mutation can also make the population exit potential local minimums and premature convergence (Siew et al., 2017, p. 10). A mutation does not happen to every individual, usually there is a predetermined chance of a mutation to happen, for example 40%.

One method of mutation is bit-flipping. In this case each individual consists of a number of bits, 0 or 1, where there is a chance that one or more random bits are flipped. If a bit is flipped, the 0 is changed to a 1 or a 1 to a 0 (Siew et al., 2017, p. 10).

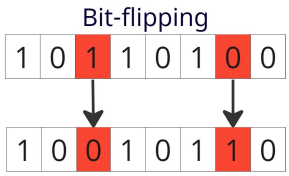

Figure 3.5: An individual gets two of its bits flipped.

Another method is by assigning the new values with a Gaussian distribution or a uniform distribution between an upper and a lower limit. When using

the first method, the new value will not often deviate too much from the original value, but will still bring something new to the gene pool. With the latter, the values will have a larger distribution, but the downside will be that it can change already good genes to something not so good. (De Falco et al., 2002, p. 286)

The mutation chance also has to be taken into account. A high mutation chance will help the algorithm to reach a higher fitness early on, but can also mutate away good solutions. An idea regarding this is to have a variable mutation rate to be able to have the best of both worlds. Here the mutation chance can be relatively high at the beginning and gradually decrease as a higher number of generations has been reached (De Falco et al., 2002, p. 287).

Chapter 4

Implementation

In this chapter it will be explained how the different applications have been made and how they are tied together. Below in figure 4.1 is it possible to see how the different parts of the thesis is built together. The code developed for this thesis can be found in appendix A.

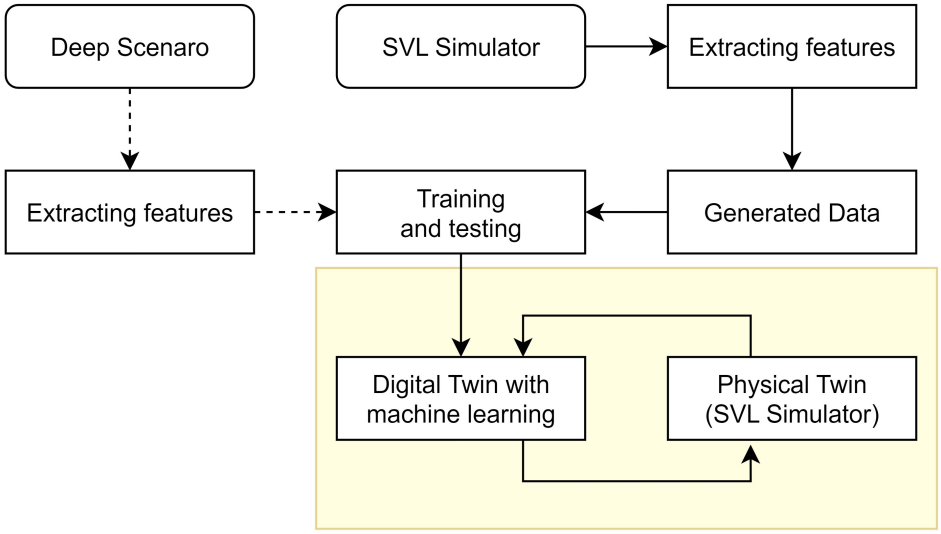

Figure 4.1: A simple overview of how the different parts are tied together.

4.1 Generating the self generated data

In this section how the self-generated data has been made. This includes how the multiple features have been collected from the simulator and how they have been processed to create new features.

To be able to create the data, the simulator was used. The *Simulation()* class was used for this as well, but it was inherited in a new class called *GenerateData()*. This class uses multiple methods from the parent class, where the primary ones are *spawnRandomNPCs()* and *calculateTTC()*. The *spawnRandomNPCs()* method can spawn a predefined number of vehicles and pedestrians. It spawns multiple different vehicles, that be it a sedan, truck, school bus or one of the other available vehicles. The pedestrians that are spawned also has multiple variations on how they look.

The heart of this class is the *generateDataWithSim* method. This method runs the simulation in the same way as the previously mentioned *Simulation.runSimulation()* in section 4.5.2. Both methods are built in the same way, i.e., configuring the needed variables and then a while-loop which runs the simulation one *updateInterval* at the time. Both methods also use the *ReadLidar()* class, but there is no prediction in this case. In this case the method writes down the values after each *sim.run()* call, namely DTO, TTC, JERK, speed and the three angular velocities. It also writes down the time used for each row in the generated data along with if the ego vehicle collided with an obstacle or not. When the time limit is reached, the while-loop is exited and the method calls another method called *storeDataGenerated()* to write down the generated values to a csv-file.

4.2 Feature engineering

Feature engineering is a crucial step in machine learning that involves transforming raw data into a format that is more suitable and informative for predictive modeling. It encompasses a variety of techniques, including selecting relevant features and creating new ones. By carefully engineering features, machine learning algorithms can better capture patterns and relationships within the data, ultimately improving the accuracy and perfor-

mance of predictive models.

This section delves into the feature engineering process implemented on the datasets to enhance the data quality. The primary objective was to augment the existing dataset by incorporating additional relevant details. For the Deep Scenario data, the information was enriched by integrating supplementary data from the corresponding scenario files 2.4.2. Similarly, to enhance collision prediction capabilities, the Self-Generated Data was augmented by rearranging its structure. This reorganization was specifically aimed at optimizing the data for improved collision prediction accuracy.

4.2.1 Deep Scenario

As mentioned in section 2.4.3, the deep scenario data consisted of several scenario files. These files needed to be combined to create a single source of data. This was done by first combining the deep scenario data generated by the different generation techniques (Lu et al., 2023) to a single csv-file.

```
1  STRATEGIES = ["greedy", "random", "rl_based"]
2
3  def Load_Files(strategy: str = "greedy") -> ...
       List[Tuple[str, str, List[str]]]:
4      return [( path, path.split("/")[-1].split("-")[1], ...
           file, strategy)
5              for path, subdir, files in ...
                   os.walk(DATASET_PATH + ...
                   f'{strategy}-strategy/')
6              for file in glob(os.path.join(path, EXT))]
7
8  def Load_Data(strategy: str = "all") -> DataFrame:
9      df = None
10     try:
11         if strategy == "all":
12             datafiles = []
13             for s in STRATEGIES:
14                 _files = Load_Files(s)
15                 print(f"Loaded {len(_files)} files for {s} ...
                       strategy")
16                 datafiles.extend(_files)
17         else:
18             ...
19
```

```
20          for path, reward, file, strat in datafiles:
21              road, scenario, _, _ = ...
                    file.split("\\")[-1].split("-")
22              f = path + "/" + file.split("\\")[-1]
23              if df is None:
24                  df = pd.read_csv(f)
25                  Add_Meta_Data(df, reward, road, strat, ...
                        scenario)
26              else:
27                  temp = pd.read_csv(f)
28                  Add_Meta_Data(temp, reward, road, strat, ...
                        scenario)
29                  df = pd.concat([df, temp], ignore_index=True)
30      except Exception as e:
31          print("Error", e)
32      finally:
33          return df
```

Listing 4.1: Loading Deep Scenario data.

As shown in listing 4.1, the data was combined by traversing the directory structure and loading all files into a list of tuples. The tuples contained the path to the file, the reward used, the file name, and the strategy used to generate the data. The list of tuples was then used to load the data into a Pandas DataFrame, with the structure seen in table 2.1:

Once this was done the newly created dataset was supplemented with additional data from the different scenario files 2.4.2. These files contained information about scenario features such as speed and angular velocity. This was firstly done by using a similar approach as when loading the raw Deep Scenario data, where the features were extracted from the XML-files and saved as a csv-file.

After this was done, the data was combined with the original dataset by using scenario id, road, reward, scenario and strategy as keys.

```
1  def ExtractAvData(self):
2      # we have 6 columns av1, ..., av6, containing a list ...
           of angular velocities (x,y,z). want to split each ...
           av into 3 columns, exp. av1x, av1y, av1z, av2x, ...
           av2y ...
3      av = ["av" + str(i) for i in range(1,7)]
4      av_ = [f"{col}{av}" for col in av for av in ["x", "y", ...
```

```
            "z"]] # av1x, av1y, av1z, av2x ...
5
6       angular_velocities = self._data[av].apply(lambda x: ...
            list(map(float, sum(map(lambda y: ...
            y.strip("[]").split(","), x.values), [])))), axis=1)
7       self._data[av_] = angular_velocities.values.tolist()
8       self._data = self._data.drop(av, axis=1)
9
10  def addFromXML(self, filename: str="") -> None:
11      """
12      Reads more data from XML files, as of now, only speeds ...
            at six different timstamps.
13
14      Params:
15          filename: str, name of file read from
16      """
17      try:
18          xmlDf = pd.read_csv(filename, index_col=0)
19      except:
20          raise FileNotFoundError(f"The file does not exist.")
21      if isinstance(self.data, pd.DataFrame):
22          self._data = self.data.merge(xmlDf, how="inner", ...
                on=["ScenarioID", "road", "reward", ...
                "scenario", "strategy"], copy=False)
23          self.ExtractAvData()
24          self.UpdateSpeedAtCollision()
25      else:
26          print("Something went wrong in 'addFromXML()'!")
```

Listing 4.2: Extrancftion of additinal scenario data.

As shown in code snippet 4.2 the data is loaded into DataFrames and merged. Additionally, the angular velocity data is extracted from a single list to three separate columns, one for each axis.

```
1   def UpdateSpeedAtCollision(self):
2       random.seed(1)
3       res = self._data.copy()
4
5       SAC = "Attribute[SAC]"
6       speeds = [f"speed{i}" for i in range(1,7)]
7       for idx, row in ...
            self._data.loc[self._data["Attribute[COL]"] == ...
            True].iterrows():
8
9           _sac = row[SAC]
```

```
10          if row[speeds[-1]] - _sac > 0:
11              continue
12
13          for speed_idx, speed in enumerate(speeds[::-1]):
14              if len(speeds) - speed_idx == 0:
15                  break
16
17              if row[speed] - _sac < 0:
18                  continue
19
20              speed_to_keep = res.loc[idx, ...
                    speeds[:len(speeds) - ...
                    speed_idx]].values.tolist().copy()
21              speed_new = []
22              last_speed = speed_to_keep[0]
23              for i in range(len(speeds) - len(speed_to_keep)):
24                  calculated_speed = last_speed + ...
                        random.uniform(-1, row[speed] / 10)
25                  if calculated_speed < 0:
26                      calculated_speed = last_speed
27                  speed_new.append(calculated_speed)
28                  last_speed = speed_new[-1]
29
30              res.loc[idx, speeds] = speed_new + speed_to_keep
31              break
32      self._data = res
```

Listing 4.3: Update speed features to use SAC as final speed.

An other important part of the feature engineering was augmenting the speed data. As shown in 4.3, the code iterates over the dataset, filtering rows that meet a specific condition. The condition requires the last speed value among the six available speed values to be greater than the SAC - Speed At Collision value.

In cases where the condition is not met, the code reverses the order of the speed values in the selected row and checks the same condition again. Once a speed value greater than the SAC value is found, that speed value, along with all preceding values, is retained and shifted towards the end of the list.

The remaining speed values are then calculated by using the first speed value from the retained values and adding a random value between -1 and the original speed value, based on the current iteration. Subsequently, the entire row is updated with the new speed values, which include the newly

calculated values as well as the values from the original row.

The modifications introduced above were necessary to address the problem of delayed collision predictions. By adjusting the speed values in the dataset, particularly by identifying the last speed value that surpasses the SAC (Speed At Collision) threshold, the code aims to improve the timing of collision predictions. This adjustment ensures that the model predicts collisions earlier, providing more accurate and timely information for collision prediction.

Lastly, some ordinary machine learning preprocessing was done. This included dropping some columns that were not needed, label encoding categorical features, scaling the data and lastly undersampling the non-collision data.

4.2.2 Self-generated data

The raw self-generated data contains no information about the previous rows when looking at each row independently. To be able to make the rows usable for a classifier this has to be fixed. It was then decided that each row should have values that have information about what happened from two seconds (four time steps) before it. Another important thing is that the rows that is noted to be a collision also have an impact on the rows that lead to that being a collision. Another important that was decided, was that a collision should have an influence on the rows before itself. As the classifier is suppose to predict a collision before it happens, the original collision row is removed to not have an impact during the training of the classifier.

After the data has been generated, it goes through another method to make it usable for training and testing the classifier. The *makeDataUsable()* method (listing 4.4) processes the data to make it usable for training. It does this by receiving the saved csv-file and loading it as a pandas DataFrame (pandas development team, 2023). It is then possible to create the new features for each row. These features then contain information about what happened before it, as seen when comparing table 3.2 with table 4.1.

To be able to collect the needed rows, the method utilizes several for-loops.

At first, the first few rows are skipped until it can have the needed rows back in time. It then iterates through the rows in the DataFrame and appends to the list behind the correct key in the dictionary (*dataDict*) that holds the data. It then appends the time for the row before it enters the second for-loop which keeps track of the row IDs that were before the current row. The row an values furthest away in time is then appended to the correct place inside the dictionary. For example if the row ID is 798, the values from row 795 to 798 is to be placed inside this entry. In this case the TTC value from row 795 is placed in TTC1, the value from row 796 is placed in TTC2 and so on. By doing this for each column, each row that is generated now has $7 \cdot 4 = 28$ columns excluding the collision ("COL") and the "toRemove" columns. Further down in the listing at lines 17-21, the collision value for the rows before a potential collision is changed to be a collision as well. The if-check at line 20 checks if there exist enough data back in time that can be changed to be a collision.

```
1  def makeDataUsable(df: DataFrame, pastImportance: int=4, ...
       rowsBeforeCol: int=5, removeCol: bool=True) -> DataFrame:
2      colsToUse = ["TTC", "DTO", "JERK", "Speed", "asX", ...
           "asY", "asZ"]
3      columns = ["Time"]
4      columns += [f"{c}{i}" for c in colsToUse for i in ...
           range(1, pastImportance+1)]
5      columns += ["COL", "toRemove"]
6      rowsToRemove = []
7      dataDict = {col: [] for col in columns}
8      for i, row in df.iterrows():
9          if row["Time"] < pastImportance-1:
10             continue
11         dataDict["Time"].append(row["Time"])
12         for j, k in enumerate(range(i-pastImportance+1, ...
               i+1), start=1):
13             for c in colsToUse:
14                 dataDict[f"{c}{j}"].append(df.iloc[k][c])
15         dataDict["COL"].append(row["COL"])
16         dataDict["toRemove"].append(row["COL"])
17         if row["COL"] == 1:
18             rowsToRemove.append(i-1)
19             available = int(row["Time"]-pastImportance+2)
20             amount = int(rowsBeforeCol) if available ≥ ...
                   rowsBeforeCol else available
21             dataDict["COL"][-amount:] = [1]*amount
22     df = DataFrame(dataDict)
23     df = df[df["toRemove"] == 0] if removeCol else df
```

```
24      df.drop("toRemove", axis=1, inplace=True)
25      return df
```

Listing 4.4: Making the generated data usable.

After the generated data has been sent through the method, the resulting DataFrame has the wanted features and can now be used as training and testing data for a classifier after it has been preprocessed. The mentioned method is also made so it is possible to change the number of rows it looks at back in time (*pastImportance*) as well as how many rows is decided to have lead to a collision (*rowsBeforeCol*).

	Time	TTC1	TTC2	TTC3	TTC4	DTO1	DTO2	DTO3	DTO4	JERK1	JERK2	JERK3
2388	12.5	22.546	11.917	13.438	15.222	22.0272	21.1773	20.4665	19.8645	2.800	2.536	0.196
2389	13.0	11.917	13.438	15.222	15.046	21.1773	20.4665	19.8645	19.2738	2.536	0.196	0.008
2390	13.5	13.438	15.222	15.046	19.378	20.4665	19.8645	19.2738	18.8359	0.196	0.008	0.008
2391	14.0	15.222	15.046	19.378	1.174	19.8645	19.2738	18.8359	13.2444	0.008	0.008	0.028
2392	14.5	15.046	19.378	1.174	0.350	19.2738	18.8359	13.2444	5.4750	0.008	0.028	0.008
2395	16.0	0.350	0.053	50.000	50.000	5.4750	0.8400	2.0090	3.3947	0.016	52.528	52.320

	Time	JERK4	Speed1	Speed2	Speed3	Speed4	asX1	asX2	asX3	asX4	asY1	asY2	asY3
2388	12.5	0.008	16.784	16.630	16.427	16.226	0.016	0.001	0.00	0.0	-0.004	0.0	0.0
2389	13.0	0.008	16.630	16.427	16.226	16.027	0.001	0.000	0.00	-0.0	0.000	0.0	0.0
2390	13.5	0.028	16.427	16.226	16.027	15.835	0.000	0.000	-0.00	0.0	0.000	0.0	0.0
2391	14.0	0.008	16.226	16.027	15.835	15.645	0.000	-0.000	0.00	-0.0	0.000	0.0	0.0
2392	14.5	0.016	16.027	15.835	15.645	15.451	-0.000	0.000	-0.00	-0.0	0.000	0.0	0.0
2395	16.0	0.920	15.451	2.125	1.879	1.863	-0.000	0.116	-0.04	-0.0	0.010	-0.1	-0.0

	Time	asX1	asX2	asX3	asX4	asY1	asY2	asY3	asY4	asZ1	asZ2	asZ3	asZ4	COL
2388	12.5	0.016	0.001	0.00	0.0	-0.004	0.0	0.0	0.00	-0.001	-0.003	-0.000	-0.0	0.0
2389	13.0	0.001	0.000	0.00	-0.0	0.000	0.0	0.0	0.00	-0.003	-0.000	-0.000	-0.0	1.0
2390	13.5	0.000	0.000	-0.00	0.0	0.000	0.0	0.0	0.00	-0.000	-0.000	-0.000	0.0	1.0
2391	14.0	0.000	-0.000	0.00	-0.0	0.000	0.0	0.0	0.00	-0.000	-0.000	0.000	0.0	1.0
2392	14.5	-0.000	0.000	-0.00	-0.0	0.000	0.0	0.0	0.01	-0.000	0.000	0.000	0.0	1.0
2395	16.0	-0.000	0.116	-0.04	-0.0	0.010	-0.1	-0.0	-0.00	0.000	0.065	-0.019	-0.0	0.0

Table 4.1: The processed data after being made into usable data. When comparing with table 3.2, notice how the time steps 15.0 and 15.5 has been removed as they are actual collision rows and that the four rows above now is a collision.

Finally a last preprocessing step was performed, including dropping some features, scaling and undersampling the non-collision data. Then only re-

maining task was to split the data into a training and test set. The training set was used to train the model, while the test set was used to evaluate the model. The test set was not used in any way during the training process, and was only used to evaluate the model after it was trained.

4.3 Implementation of the features

As discussed in section 2.4.2, various features are utilized in this thesis. This section focuses on how these features are collected from the physical twin and utilized. Specifically, the *time to collision* (TTC) and *distance to obstacle* (DTO) features play a significant role in this thesis and are extensively calculated and utilized. While the Deep Scenario documentation (Lu et al., 2023) provides a description of what these features are, it offers limited information on how they are determined. The following section will elaborate on the methodology employed to derive these features.

4.3.1 Distance to obstacle

The attribute DTO is as mentioned earlier a feature that says something about the distance to another obstacle. It is not obvious from Deep Scenario if this is measured in a general direction or if it is just in front the vehicle, another uncertain thing is the feature itself, but it is assumed it is in meters.

During the development of the digital twin, the initial approach to obtain the Distance to Obstacle (DTO) from the ego vehicle involved collecting the x, y, and z coordinates of all vehicles and pedestrians within the simulation. To be able to calculate the distances between the ego vehicle and all other potential obstacles the formula for Euclidean distance were used. The core of this formula is the Pythagoras theorem:

$$c^2 = a^2 + b^2 \tag{4.1}$$

When calculating the Euclidean distance, the a and b part gets exchanged with the distances along the x-coordinate and the y-coordinate. Here the z-

coordinate is not mentioned because this is for a two-dimensional distance. The exact same formula is used when there are three dimensions, it is just to add the difference between the z-coordinates and square it. This then ends at the formula:

$$\begin{aligned} c^2 &= (x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2 \\ c &= \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2} \end{aligned} \tag{4.2}$$

The positive part about this method is that it is fast to calculate and it gets the distance to all NPCs in the simulation. The downside is that is does not register buildings, trees, poles and other potential obstacles. Another thing is that is dependent on information from the simulation itself, which is a bad thing if this were to be used in a real life scenario, it would simply not get the coordinates of anything.

The solution was to use one of the many sensors on the ego vehicle. More specifically the LiDAR sensor. There are also other sensors on the vehicle that maybe could have been used, like the radar, but these were not easily available within the Python API.

The type of LiDAR the ego vehicle possesses, works as mentioned in section 2.3.3. The sensor have 32 lasers aligned vertically and each measures 360 times per rotation. This results in a horizontal field of 41.3 degrees, which means that the angle between each laser is around 1.3 degrees. The total resolution of the point cloud will then be $32 \cdot 360 = 11520$ points. The rotation frequency is 10 Hz. In the simulator the resulting point cloud can be stored as a pcd-file which then later can be used by a program. The maximum distance this LiDAR can measure is 100 meters and the closest is 0.5 meters.

To be able to calculate the DTO, the point cloud from the LiDAR sensor is used. One point cloud is saved every so often, so it keeps getting an updated version to use and process. To be able to process the pcd-file there is a Python library called Open3D (Zhou et al., 2018). This library can be used to load and visualize a point cloud, as seen in figure 2.8. It can also load the point cloud as an array, which then can be easily processed by some code. This array consists of all the points from the point cloud

where each point has an x-, y-, and z-coordinate where the origin of that coordinate system is the LiDAR sensor itself, which is placed on top of the ego vehicle. The coordinate system is a right handed system, where the x-axes is going forwards, the y-axes is going to the left and the z-axis is going up, all in respect to the orientation of the vehicle. The points in the array is then stored after each other, so all points the most bottom laser produces in one rotation is indexed first in the array, then the next 360 points from the laser above is indexed and so on. The result is one array consisting of up to 11520 smaller arrays that contain the x-, y-, and z-coordinates.

Since DTO is a single measure, it is assumed to apply only to obstacles in front of the ego vehicle. This is because the ego vehicle is usually moving, and it is easier to handle and compute potential collisions with objects ahead of it rather than those on its sides, as seen in figure 4.3

The first problem was to figure out which points were in front of the vehicle. Because each laser produces 360 points per rotation and the following points from the next lasers are stored after each other, it is possible to iterate through the array and only keep those points in front of the vehicle by looking at the indexes. The indexes should be corresponding to one degree difference horizontal, and when the 360th point is reached, the next 360 points are from a laser that is aiming some degrees further up and so on until it reaches the 32nd laser, as seen on the "circles" in figure 4.2.

To be able to not only look at the one degree straight in front, it was first tried to also look at n points from each laser. This resulted in horizontal field of view of $1.3n$ degrees.

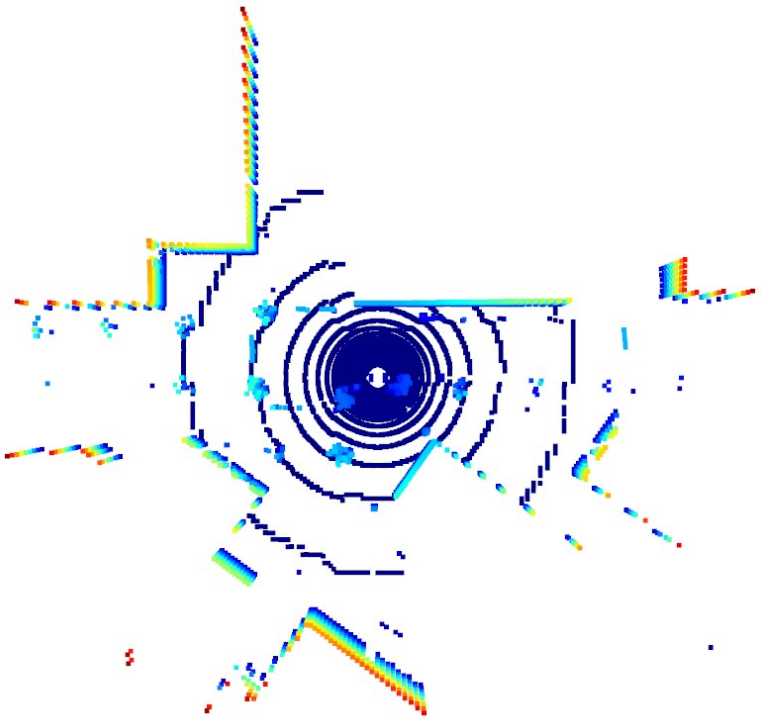

Figure 4.2: A top down view of the point cloud, the ego vehicles is at the center. Each "circle" is from a different laser looking 1.3 degrees further up. The direction of the ego vehicle is to the right.

The next problem was to figure out what each point is, whether it represented an obstacle or the ground. Since the origin of the coordinate system is the LiDAR itself, which as mentioned is placed on the roof, the ground the that the vehicle drives on has a constant negative z-coordinate of approximately *-2.30* when driving on a flat surface. This means that everything that has a z-coordinate value that is higher than that can potentially be collided with. Points that have a positive z-values will then be at the same height as the sensor, thus it is assumed the if the value is above *0* is not going to collide with the ego vehicle. If the vehicle is accelerating, the car is going to be tilted either up or down. This will affect the coordinate system as the car rotates slightly. Now the z-coordinate is no longer the constant *-2.30*, but it will be a flat plane that is non parallel with the x and y plane. To compensate for this it is either possible to calculate what this plane is

by finding the points that are corresponding to the ground, or to make the z-value that should be the ground to have a higher constant. The latter of these two alternatives are chosen in this thesis, as it is easier to implement and this also has a side effect that it does not register small bumps in the road or other objects that can be driven over.

Another thing is that since the LiDAR is placed approximately at the center of the vehicle, the distance to an obstacle in front is going to be the distance between the front bumper of the car and the distance between the LiDAR and the front bumper. Because of this, the latter distance needs to be figured out and subtracted from the distance from the LiDAR to an obstacle to find the correct DTO. This distance was figured out by running the simulator and colliding with a building, in that way it was possible to use the LiDAR to get the distance to the building, which also would be approximately the length between the sensor and the vehicle's bumper. By doing this, it is possible to be very close to the exact distance to an obstacle that is in front by subtracting this value from the x-coordinate. The distance to the bumper was measured to be *2.87*.

When driving around, it was noticed some weird DTO values. By manually checking the corresponding point cloud, it was notices that it used values that it should not have used. Some points were scattered around the vehicle in all directions, when it should have only seen points that are directly in front of the vehicle. The solution to this was to change the way these points were found. Instead of looking at the indexes of the points, it was looked at the coordinates themselves. This was a viable solution because the y-axis goes directly in front of the vehicle. By checking if the y-coordinate was above or below a certain value, it could be determined if that point could be something that it is possible to collide with.

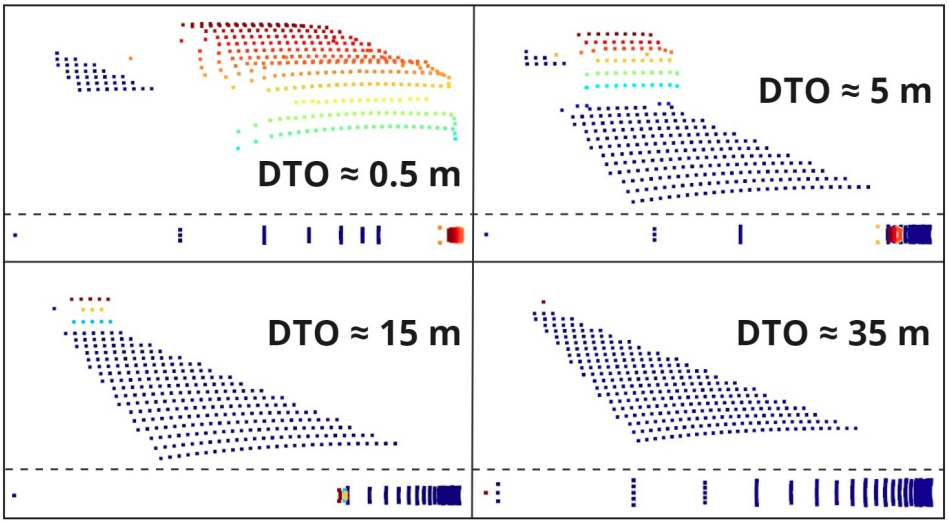

Figure 4.3: What the LiDAR sees when the same NPC vehicle is located at different distances from the ego vehicle. The bottom part of each section is the same point cloud from a top down view.

In the implementation of the DTO, only the coordinates that fall within a threshold is looked at. The x-coordinates has to be greater than the length of the distance from the LiDAR to the front bumper, 2.30. The y-coordinate has to be greater then -1.0 and less then 1.0. In this case, the LiDAR scans the area that is in front of the vehicle and within the width of the vehicle. It also take into account that the z-coordinate is greater than -1.9. This means that the blue points in figure 4.3 are not taken into consideration, but every other point is. The resulting DTO will then be as noted in the figure. When there is no such points in the point cloud, the resulting DTO will be 100. 100 meters is also the maximum distance the LiDAR can measure.

4.3.2 Time to collision

The next attribute to figure out was the *time to collision* (TTC). In the documentations this is as mentioned a value that says something about the time it would take for a collision to happen. The scenario here is that the ego vehicle is driving forwards and has a direction towards an obstacle and is going to collide with this obstacle if no action is taken, that is that

either the ego is turning or decelerating or that the obstacles moves out of the way. To calculate this, the main parameters are the DTO and the speed of both the ego vehicle and the obstacle. It is also possible to use the acceleration of both of these objects as well, but it was determined that this was not important as the function assumed that the accelerations were constant. Because of this, the function could say that the vehicles would reach a speed that it would not realistically reach, for example that it would start going backwards or reach a very high speed eventually. It could be possible to create a function that would take this into consideration, but it was deemed as not necessary.

To calculate the time it would take to collide, the first parameters are already collected, but the obstacles speed is more of a challenge to get. To get the obstacle's speed one could simply get the speed from the simulator, but this would not be possible in a real life scenario, so the speed has to be gotten from somewhere else.

To be able to calculate the obstacle's speed, the obstacle's change in displacement is needed, and to get that, two pairs of the DTO and the ego vehicle's speed are needed, as well as the time interval between the pairs. When these are collected, equation 4.3 can be used to get the displacement d_{obs}, and this again can be used together with the time interval to get the speed, as seen in equation 4.4 below. Together the obstacle's speed can be obtained with formula 4.5. Here the ego vehicles speed, is the average of the speed when the two distances are measured.

$$
\begin{aligned}
d_{ego} &= t \cdot \frac{v_1 + v_0}{2} \\
d_{obs} &= d_{ego} + dto_1 - dto_0 && (4.3) \\
v_{obs} &= \frac{d_{obs}}{t} && (4.4) \\
v_{obs} &= \left(t \cdot \frac{v_1 + v_0}{2} + d_1 - d_0 \right) \cdot t^{-1} && (4.5) \\
&= \frac{v_1 + v_0}{2} + \frac{d_1 - d_0}{t} && (4.6)
\end{aligned}
$$

The next step is to use one of the functions of motion (equation 4.7) to derive a function (equation 4.8) that can be used to get the TTC value. At

first the acceleration was also used as a way to get more information into the function, but as the DTO is quite large sometimes, it would influence the speed of the vehicle in the function to be either unreasonable high or low as the acceleration is not constant at all speeds. Thus, it was more accurate to not use it in the calculation.

$$
\begin{aligned}
d &= vt + \frac{1}{2}at^2,\ a = 0 && (4.7)\\
d &= vt \\
\hookrightarrow t &= \frac{d}{v} \\
\longrightarrow t_{TTC} &= \frac{d}{v} = \frac{d_{obs}}{v_{rel}},\ v_{rel} = v_{ego} - v_{obs} && (4.8)
\end{aligned}
$$

The resulting formula in equation 4.8 is using the relative speed difference between the ego vehicle and the obstacle, that is because it only needs to calculate the time it uses to catch up to the obstacle.

After using the formulas above, it was noticed eventually that the calculation could and should be shortened. The calculation is shown below in equation 4.9.

$$
\begin{aligned}
t_{TTC} &= \frac{d_{obs}}{v_{ego} - v_{obs}} \\
&\text{Substitute in for } v_{ego} \text{ and } v_{obs} \\
t_{TTC} &= \frac{d_1}{\frac{v_1+v_0}{2} - (\frac{v_1+v_0}{2} + \frac{d_1-d_0}{t})} \\
&= t \cdot \frac{d_1}{d_0 - d_1}
\end{aligned} \qquad (4.9)
$$

As a result, the formula is more precise as it now do not need many of the previously used parameters as they cancel each other out. If the resulting value is above 50 or if d_1 is higher then d_0, the resulting TTC will have a value of 50.

4.3.3 Jerk

Jerk is a measure that says something about the comfort for the passengers. When looking up what jerk is, it comes up as the change in acceleration and it's metric is m/s^3. Jerk is a relatively unknown and unused measure. Jerk is often experienced as uncomfortable, for instance when the driver of a car steppes on the brakes and when the brakes are suddenly released. Jerk is one of the derivatives of change in position and time. The more common measures derived from this is velocity (m/s) and acceleration (m/s^2), where velocity is the first derivative and acceleration is the second derivative. Jerk is then the third derivative. There are also more derivatives like snap, crackle and pop. (Eager et al., 2016)

To calculate jerk from the acceleration values, it is needed the currents acceleration value and the most previous one. The formula will then be as follows in equation 4.10:

$$jerk = \frac{a_1 - a_0}{t} \tag{4.10}$$

From this formula, jerk can both be positive and negative, but by looking at the values in the Deep Scenario dataset, jerk is a value that is always positive. It is then assumed that one can just take the absolute value of it and use that.

4.4 Implementation of the model

This section focuses on the essential steps undertaken in preparing the data and implementing the collision predicting models. The implementation details of the preprocessing steps, such as label encoding, feature selection, and data scaling, are explored. Additionally, the chapter delves into the machine learning aspects of this thesis, and specifically the implementation of them.

4.4.1 Preprocessing

As discussed in the section on feature engineering (section 4.2), a series of essential preprocessing steps were conducted to prepare the data for model training and testing. These steps encompassed various tasks, such as label encoding categorical features, feature selection by dropping certain variables, data scaling, and under-sampling the non-collision instances. This section will delve into the implementation details of these preprocessing techniques, highlighting their significance in preparing the data for subsequent analysis and model development.

```
1  # undersample
2  self._data = ...
       pd.concat([self._data[self._data["Attribute[COL]"] == ...
       False].sample(sampleSize, random_state=1), ...
       self._data[self._data["Attribute[COL]"] == True]])
3  ...
4  def preProcess(x, y):
5      ...
6      x[cols] = x[cols].apply(LabelEncoder().fit_transform)
7      ...
8      y.replace(False, 0, inplace=True)
9      y.replace(True, 1, inplace=True)
10
11     for c in x.columns:
12         if x[c].dtype != float:
13             x = x.drop(c, axis=1)
14     ...
15     if not self._fitScaler:
16         self.scaler.fit(x)
17         self._fitScaler = True
18     x = self.scaler.transform(x) # Scaling the data
19     ...
20     return x, y
21 ...
```

Listing 4.5: Condensed preprocessing implementation

The provided code listing 4.5 showcases a shortened version of the preprocessing method used in this thesis. The *preProcess()* method performs several preprocessing steps.

Firstly, an undersampling technique applied to the dataset. By selectively

extracting a subset of data, the code ensures a balanced representation of non-collision and collision instances. Specifically, the code combines a randomly sampled portion of non-collision instances with all the available collision instances, creating a balanced dataset for further analysis and modeling.

The initial step of the preprocessing method involves label encoding of selected columns in the dataset. By applying the *LabelEncoder()* function from the scikit-learn library (Pedregosa et al., 2011), the categorical data in the specified columns is transformed into numerical representations, making it easier for machine learning algorithms to process.

Next, the code removes non-numeric columns from the dataset. This step ensures that the data consists only of numeric features, which is often a requirement for many machine learning algorithms. Any column with a non-float data type is dropped from the dataset, thus retaining only numeric features for subsequent analysis.

Then finally some feature scaling of the data is performed. This ensures that the data have zero mean and unit variance, which usually is crucial for many machine learning algorithms.

4.4.2 Machine Learning Models

As mentioned in section 3.2.3 the machine learning models employed where MLP Classfier, Random Forest Classifier, SVM Classifier and XGBoost Classifier. All of these models were implemented using the scikit-learn library Pedregosa et al. (2011). Scikit-learn is a popular Python library for machine learning. It provides a wide range of tools and algorithms for various tasks, including classification, regression, clustering, and dimensionality reduction. With its user-friendly and consistent API, scikit-learn simplifies the process of implementing machine learning models and evaluating their performance.

Besides these models, a wrapper class was implemented which had the MLP classifier as a base model. This class had the model which was going to be used as a parameter, which made it easy to switch between models. This class also had some additional functionality, such as the ability to do

some preprocessing, save and load the model, as well as some other general functionality.

```
1  # Random Forest Classifier
2  rf_clf = ...
       DTPredictor(RandomForestClassifier(n_estimators=100, ...
       max_depth=10, random_state=1))
3  rf_trainX, rf_trainY = rf_clf.preProcess(trainX.copy(), ...
       trainY.copy())
4  rf_testX, rf_testY = rf_clf.preProcess(testX.copy(), ...
       testY.copy())
5
6  rf_clf.fit(rf_trainX, rf_trainY)
7
8  rf_pred = rf_clf.predict(rf_testX)
9  rf_score = rf_clf.getScore(rf_testY, rf_pred)
10
11 model_score = f"{rf_score[0][0]}-{rf_score[0][1]}-..."
12 rf_clf.saveModel(f"RandomForestClassifier_{MODEL_PREFIX}", ...
       accuracy=model_score)
```

Listing 4.6: Model implementation structure

Now training, evaluating and saving the model was as simple as, seen in listing 4.6. The wrapper class *DTPredictor()* takes in the model as a parameter, then does some preprocessing, trains the model and evaluates its performance. Lastly the model is saved using these scores as a part of the name. The same was done for each of the other models, making it easy to quickly try out different models.

4.5 Using the simulator

In this section it will be explained how to use the simulator and also how to create and implement both of the digital and physical twin.

4.5.1 Getting started

One of the first objects of this thesis was to run the SVL Simulator. To be able to do so, one have to download the simulator from OSSDC's *GITHUB page*[1]. The next step is to follow the steps in the *Python API documentation* to be able to use the simulator through the API.

4.5.2 Creation of the twins

To be able to crate the physical twin the *Simulation* class has to be made. This class is going to launch the simulation, set the parameters for the environment, load the map, spawn the NPCs and of course the physical twin, also known as the ego vehicle. It is also going to calculate and utilize other classes to calculate the correct values for the multiple features.

Initialization

On initialization, the class sets up the environment and connects to the simulator through the correct ports. It also loads in the map and spawns the ego vehicle. Following the initialization it is possible to use the *runSimulation()* method. This method starts the simulation, but it also sets the needed parameters that are going to be used. The most important input parameters to the method are *simDuration*, how long the simulation us going to last, *updateInterval*, how often the digital twin receives updates, *window*, how many meters to the left/right the LiDAR is looking for obstacles and *model*, specifying the machine learning model (see section 3.2.3 the digital twin is going to use for its collision detection. Another useful parameter is *scenario*, this decides if the simulation is going to run with one of the predefined scenarios or if the ego vehicle is going to be controlled by a keyboard.

```
1 from environs import Env
2 import lgsvl
3 class Simulation():
```

[1]OSSDC's GitHub page: `https://github.com/OSSDC/OSSDC-SIM`

```
4       def __init__(self, map: str="bg") -> None:
5           self.env = Env()
6           self.sim = lgsvl.Simulator(
7               self.env.str("LGSVL__SIMULATOR_HOST",
8               lgsvl.wise.SimulatorSettings.simulator_host),
9               self.env.int("LGSVL__SIMULATOR_PORT",
10                  lgsvl.wise.SimulatorSettings.simulator_port)
11              )
12          ...
13          self.ego = self.sim.add_agent(
14              self.env.str("LGSVL__VEHICLE_0",
15              lgsvl.wise.DefaultAssets.
16                  ...ego_lincoln2017mkz_apollo5),
17              lgsvl.AgentType.EGO, self.state
18              )
19      ...
```

Listing 4.7: Initialization of Simulation(), found in usingSim.py.

Inside the mentioned *runSimulation()* a lot is going on. The first thing that happens is to load the correct model with the *Predicter()* class (either the one made for the data from Deep Scenario or for the self generated data). Following that the *ReadLidar()* class is initialized with the correct parameters.

Getting the features

After the needed variables and classes are loaded, the simulation can begin. This happens inside a while loop which starts by calling *self.sim.run(updateInterval)*. This will run the simulation for the duration of *(updateInterval*, before it will continue down the loop to read the currents speed of the ego vehicle and calculate the needed features. Among those are DTO and TTC. To be able to calculate both of those feature, the LiDAR has to be used. This happens in the mentioned *ReadLidar()* class. Before that class can do what it does, the point cloud from the LiDAR sensor has to collected, this can be seen in listing 4.8 below.

```
1  class Simulation():
2      ...
3      def runSimulation(self, ...)
4          ...
```

```
5           lidar = ReadLidar(window, 35)
6           ...
7           while True:
8               self.sim.run(updateInterval)
9               self.ego.get_sensors()[2].save(PATH + ...
                    "/data/lidarUpdate.pcd")
10              dtoList.append(lidar.updatedDTO)
11              ...
```

Listing 4.8: A part of runSimulation()

After the point cloud is saved, the mentioned class can do what is needed. Inside the class, the point cloud is loaded when the property *updatedDTO* is called. After the loading, the class can figure out the distance to a potential obstacle in front of the ego vehicle, as explained in section 4.3.1.

Following this, the next steps are to collect the speed of the ego vehicle. When this is done, the rest of the features can be calculated with what is now collected, namely acceleration, jerk and TTC. Then the three angular velocities are collected

Predicting

The next step is to start predicting. The loaded classifier is either trained with the data from Deep Scenario or with the generated data. Since the features used are different for the two, this is accounted for so the correct one is used. Before a prediction is made, the features are preprocessed in the same matter as the trained data and is then used to predict a collision. If the classifier predicts a collision, the digital twin tells the physical one to apply the brakes and turn on the hazard warning lights. After a couple of seconds, the brakes are lifted and the vehicle can continue to drive.

Visualizing the scenario

There are also an additional method, namely *plotting()*. This method is plotting how a simulation went and its development (see figure 4.4). This method is also able to store the parameters used in a csv-file, so the data

can be looked at at a later time, which allows for measuring how well the classifier did on an actual driving scenario at real time.

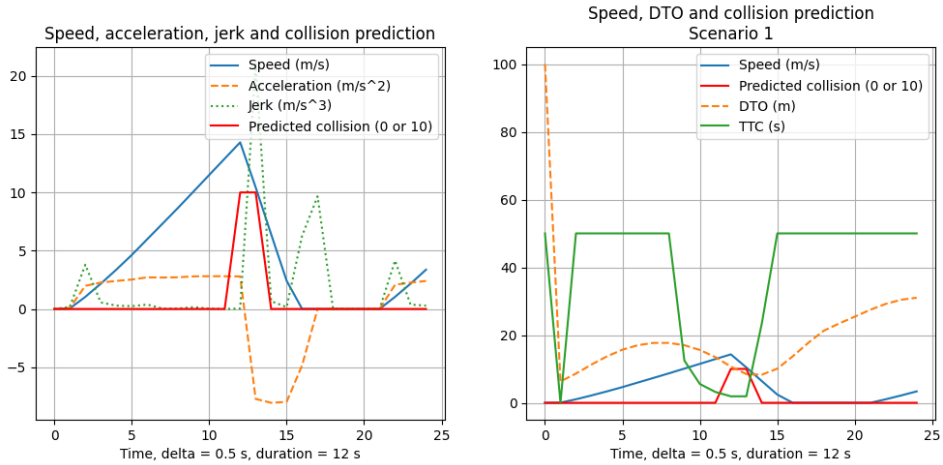

Figure 4.4: The plot generated after running scenario 1 with the XGB classifier as the brains of the digital twin. In this scenario, the ego vehicle stopped right before colliding with another vehicle. The scenario is explained in section 5.1.

The moment the ego vehicle has predicted a collision and the digital twin has told its physical counterpart to apply the brakes, can be seen below in figure 4.5.

Figure 4.5: The ego vehicle manages to stop in time when the digital twin tells the physical twin to apply the brakes.

4.6 Implementation of the genetic algorithm

In the implementation of the genetic algorithm, several iterations of each method were created, but not all of them gave results that were usable. In this implementation, a class called *GeneticAlgorithm()* is created and it has several different input parameters; *populationLimit, maxGenerations, minValues, numberOfVariables, tmParticipants, variation, mutationChance, fitnessGoal, toKeep*. These will be explained below together with how it is made.

4.6.1 Population

At initialization, the parameters are set and the starting population is generated. Both the parameters *populationLimit* and *numberOfVariables* are integers, where the first one decides how many individuals each generation consists of and the other one is the amount of genes per individual. The other two parameters are lists that determine the minimum and maximum values each gene can have, i.e., the *nth* gene has to be a value the is between the *nth* element of *minValues* and the *nth* element of *maxValues*. By popu-

lating the individuals in this way, each individual consists of genes that may be found within the actual dataset. In this case, the genetic algorithm is made to have 28 genes per individual and a total of 1000 populations which produces up to 10,000 generations.

4.6.2 Fitness

The next step is to figure out the fitness of the population. Here each individual is sent through the fitness function to get a score. This score is calculated by:

$$fitness = \frac{1}{\sum_{i=1}^{4} abs\left(ttc_i \cdot speed_i - dto_i + \frac{ttc_i^2}{dto_i} - \frac{speed_i}{2} \cdot w_i\right)} \quad \text{where } w = 0.1, 0.15, 0.25, 0.5 \tag{4.11}$$

By looking at the formula (equation 4.11), the closer the bottom part is to zero, the higher the fitness will be. There is also a check which makes sure that there will be no division by zero error when running the code. If the bottom part is zero, the resulting fitness is determined to be very high.

There is also the class parameter *fitnessGoal*, which is a relatively large integer. This parameter makes the algorithm return the current generation if the most fit individual has a fitness that is higher than the mentioned parameter.

The goal of the fitness function is to find the best individuals, and it does this by the mentioned fitness function. The function is though out by knowing that the TTC, DTO and speed features have a high impact on whether a collision is predicted or not. After knowing this, the part with $ttc \cdot -dto$ is suppose to be zero, as time multiplied by speed should equal distance. The other part with $\frac{ttc^2}{dto} - \frac{speed}{2}$ is more derived from testing. Finally, this part of the equation is multiplied by a weight. This is to give the older values a lower impact on the final score.

4.6.3 Selection

In the selection, two parents are chosen by a tournament selection as described in section 3.3.2 and the distribution between the number of participants can be seen in figure 3.3. In this case, the number of participants is chosen by the integer class parameter *tmParticipants*.

There is also a check if the two parents are diverse enough to each other. This check is made to look at the *nth* gene of each parent and finding out if they are within 0.01 of each other. If there exists one such gene, the parents are to be picked again. This function is also recursive.

Since there is a small chance that the best performing individuals are not chosen during the selection process, the integer class parameter *toKeep* can ensure that at least some are. It does this by taking the *toKeep* most fit individuals and placing them straight into the next generation to force those genes to be a part of the next generation.

4.6.4 Crossover

In the crossover function, the genes from the two parents are mixed to make two children. The genes of the parents are mixed in a uniform distribution, as seen in figure 3.4 under the *uniform* headline and explained in section 3.3.2.

4.6.5 Mutation

After a child has been produced, there is a certain chance that the child is mutated, this chance is given by the class parameter *mutationChance*. This parameter is a floating number between 0 and 1. If a child is to be mutated, an average one fourth of these genes are mutated. If a gene is mutated, its new value is going to be a chosen at random to be within a range of its old value plus and minus *variation*. If either one of the end points in the range are outside of the minimum or maximum value determined in *minValues* or *maxValues*, the new value is either the minimum or maximum value.

4.6.6 Running the algorithm

When all the needed methods have been made, the algorithm can start. The *run()* method starts by a for-loop that can go for up to *maxGenerations*. Then inside, the method sends the current generation through the fitness function to get a score, the generation is then sorted in an ascending order based on the fitness. If the most fit individual is greater than *fitnessGoal* the algorithm returns the sorted list of the current generation and the individuals fitness score.

If the algorithm does not exit, the reproduction of the next generation starts. First the *toKeep* most fit individuals is sent through, then the algorithm makes use of the selection, crossover and mutation methods to produce children until the *populationLimit* is reached.

When running, the algorithm also prints to the console the best individual for every 100th generation as well as its score. When finished, the algorithm returns the same sorted list and also prints the best individual to the console.

Chapter 5

Results and discussion

In this chapter, the implementation is tested out and a result is given. This is for digital twins trained with data from either Deep Scenario or the self-generated one. The genetic search algorithm is also tested. The results are also discussed and reasoned around. Finally the research questioned are answered.

5.1 Digital twin

To measure if the created digital twins are any good, each model has been sent through the same scenario. The models were also evaluated during the training and testing period. In that period each model that is based on the data from Deep Scenario has the same test, train split from the data. The same can also be said about the models based on the self-generated data. Below in table 5.1, the confusion matrix of each model can be seen.

	TN	FP	Models							
	FN	TP	MLPC		RFC		SVMC		XBGC	
Datasets	DeepScenario		569	24	582	11	577	16	576	17
			28	189	32	185	89	128	24	193
	Self-Generated		75	34	82	34	82	38	82	15
			22	68	15	68	15	64	27	75

Table 5.1: Confusion matrix for each model that is either trained with data from Deep Scenario of the self-generated one. The values are from predicting with the test data from each dataset. Top left represents the true negatives, top right represents the false positives, bottom left represents the false negatives and the bottom right represents the true positives.

The evaluation of the classifiers, as discussed in section 3.2.4, plays a crucial role in assessing their performance using key metrics such as accuracy, precision, recall, and AUC. These metrics provide valuable insights into the models' predictive capabilities and overall effectiveness. However, it is equally important to bridge the gap between the metrics and real-world performance by considering the models' behavior in driving scenarios within the simulator. During our evaluation, we observed notable differences between the metrics and the models' actual performance.

For instance, the MLP classifier demonstrates a high accuracy of 0.94 and precision of 0.89, indicating overall strong performance. However, upon examining the driving results, it becomes evident that the MLP classifier fails to classify collisions early enough, resulting in collisions.

Similarly, the Random Forest classifier demonstrates a high accuracy of 0.9469 and precision of 0.9439, indicating its overall effectiveness. However, when assessing its performance in the driving simulator, we observed instances where the model still resulted in collisions.

In contrast, the SVM classifier, despite having a comparatively lower AUC score of 0.78 and other metrics, demonstrated the best performance in the driving simulator. Although its accuracy of 0.8704 and precision of 0.8889 were not the highest among the evaluated models, the SVM classifier consistently predicted collisions earlier than other models. This proactive approach potentially allowed for more time to initiate braking or other preventive measures, leading to a higher chance of collision avoidance.

So to be able to compare all the different models, a common scenario all can be evaluated on had to be created. This scenario is actually a combination of two scenarios. In the first one, the ego vehicle is starting around 5 meters behind an NPC vehicle going 8 m/s and the throttle is set to 0.5 on the ego vehicle. This means that at the first seconds, the NPC is driving away from the ego, before the ego eventually caught up to it because is drives faster. In the second scenario, an NPC vehicle is spawned 95 meters in front of the ego vehicle. The ego then accelerates with full throttle until a speed of approximately 18.5 m/s is reached. It then coasts until it collide with the NPC. These two scenarios are then combined into one, where the splice is directly after the collision of scenario 1. The DTO and when the collisions happened in the scenario can be seen at the top two rows of both figure 5.1 and in figure 5.2. Also note that the DTO starts at around 90 before it goes up to 100 and then goes down to 60. In reality, the vehicle were always closing in on the NPC. This happens because the LiDAR sensor do not have a high enough resolution to cover the area the NPC is in at these time steps. In other words, the LiDAR has a blind spot somewhere between 60 and 80 meters away. If the vehicle were taller, this might not have happened.

Trained with data from Deep Scenario

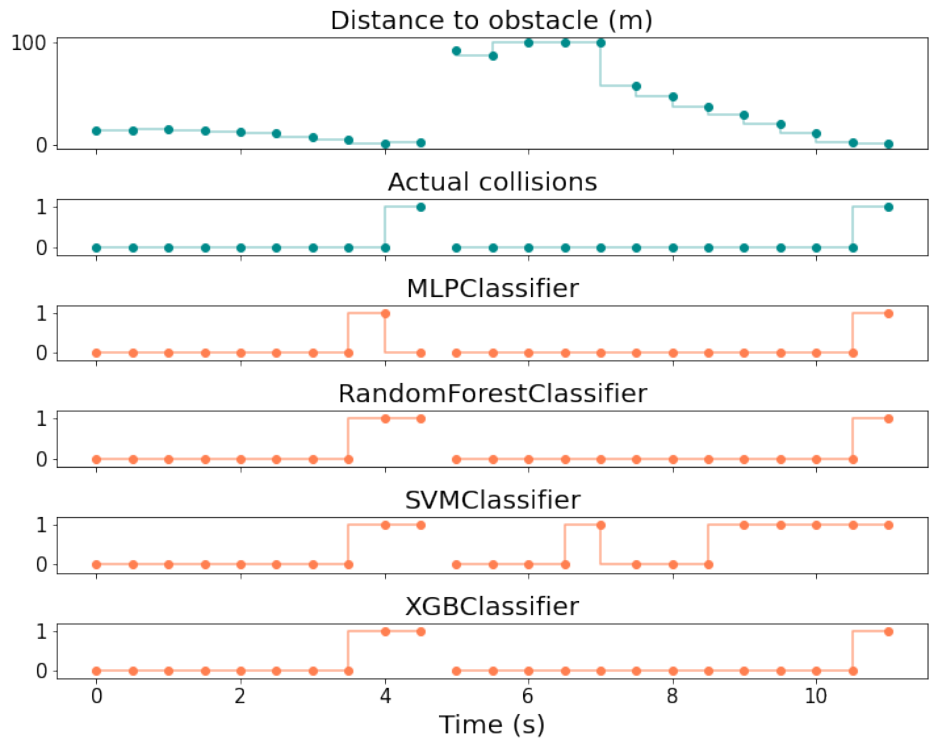

Figure 5.1: Comparing the prediction collision with the actual collisions. Here each model has been trained with data from the Deep Scenario dataset. 1 represents a collision prediction and 0 for not a collision.

In figure 5.1 above, the different classifiers are trained with data from the Deep Scenario dataset. By looking at the different classifiers for scenario 1, all of them starts predicting at the same time, but the difference is that the MLP classifier do not believe that the actual collision is a collision itself, whereas the other ones does. The data they are trained on, only has one DTO and JERK value and six speed values. It is assumed that the collision happened after the last speed value, but there are something strange regarding when the collision happens, as mentioned in section 4.2.1.

For the second scenario, MLP, Random Forest and XGB predicted only that the last point is a collision, which is the truth, but the vehicle would

not manage to brake and avoid the collision if they were used. The SVM classifier predicts a collision quite early, and then changes its mind before a more appropriate prediction is made. If the brakes were applied at the first prediction, it would have braked to early and the latter prediction would probably be okay. When comparing the result with the confusion matrix in table 5.1, it is easy to see that the SVM classifier would predict before an actual collision, as its number of false negatives is quite high.

By looking at the results, the Random Forest and the XGB classifier are the most true to the actual collisions. The reason for why they are predicting the same collisions might be because they are both based on decision trees. The classifier with the highest chance of avoiding a collision in this scenario would be the SVM classifier.

Trained with the self-generated data

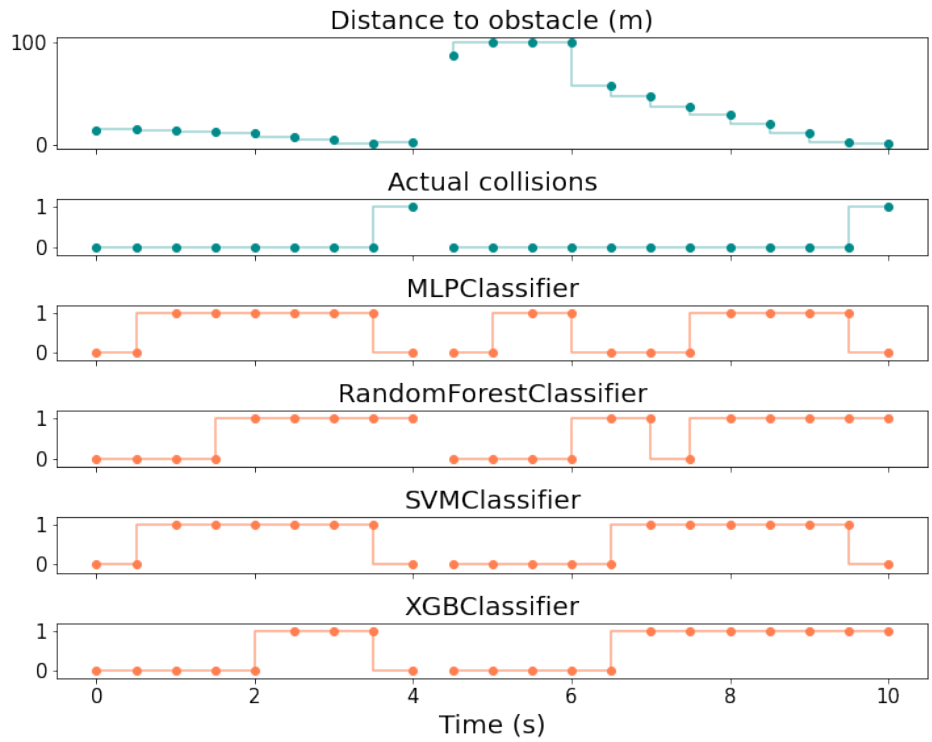

Figure 5.2: Comparing the prediction collision with the actual collisions. Here each model has been trained with the self-generated data. 1 represents a collision prediction and 0 for not a collision.

In the figure above (figure 5.2), the different classifiers are trained with data from the self-generated dataset. Something to remember here is that all these classifiers are trained with data that have four time steps for each feature (*pastImportance*=4), the *rowsBeforeCol* parameter is set to 5 and then the collision row is removed from the training data (see section 4.2.2 for more information).

By looking at the results in the figure, all the classifiers are predicting that a collision is going to happen for the first scenario, the difference is when. Both the MLP classifier and the Random Forest classifier predicts that a collision will happen quite early. If this was a real scenario, the vehicle

would stop way too soon. The other two models predicts that a collision will happen a bit later, end would end up stopping closer to the NPC vehicle if the brakes were applied. A more detailed plot of the scenario with the XGB classifier can be seen in figure 4.4, here the brakes were actually applied. In the second scenario, the MLP and the Random Forest classifiers predicts that a collision is going to happen. Regarding the first one, it predicts a 1 (collision) when the DTO is still high, which is a strange behaviour. For the latter classifier, it predicts a collision when in the moment the DTO drops, this also has a direct impact on the TTC value. In this moment it could be possible that the classifier believes that a vehicle is coming towards it with a high speed. Also remember the LiDAR's blind spot mentioned earlier, so this might not have been predicted if the LiDAR resolution was higher. About the SVM and the XGB classifiers, they start to predict collisions at the same time in this scenario. Since the speed is higher in this scenario, applying the brakes early might be the right call. The other two classifiers might be just correct at their second predictions, but it could be very close to colliding with the NPC.

Additional comments

As seen in the figures and by driving manually with a loaded classifier in the simulator, the prediction is not always the best. There could be many reasons for this, that be it the LiDAR's blind spot, too few features, for instance, the LiDAR should potentially have a wider field of view and account for the ego vehicle's direction, for instance while turning. The classifiers for digital twins could also have had more training data to account for more scenarios the vehicle can find itself in. Another note about the LiDAR is the maximum distance it can measure, which is only 100 meters. This can be a rather large issue when driving at high speeds, especially when the braking distance can reach such lengths.

Modern cars also have many different sensors, and so do the ego vehicle. It has the potential to use them to generate more data, which can lead to a better understanding for what is going to happen. One can think that there is a reason that they are there.

Something else to mention, is that the collisions should maybe have been a regression task and not a classification. The self-generated data could have

had a percentage of risk, so for example if the vehicle has a speed of 5 m/s, it has a more time to process the environment than if it had a speed of 20 m/s. In that case the chance of a collision is higher, just because of the less time the digital twin has to react in case of a car suddenly decides to stop, drive onto a road or another something else a car might do. If the ego vehicle scans the environment as well, it might find the higher speed to be acceptable. The vehicle could for instance be on a tight city road where it should have a lower speed to account for unforeseen circumstances that could end in a collision. The difficult part regarding this, could be how to determine the collision value.

By looking at the results, the best digital twin out of these, would be the one using the XGB classifier to predict collisions that is trained with the self-generated data.

5.2 Genetic algorithm

The goal of the genetic search algorithm is as mentioned in research question 2, to create testing data for the digital twin. All the generated data is suppose to be a row that can lead to a collision. To evaluate this, the data was sent through to the digital twin and the data was also compared to actual collision data to find out how realistic the data is.

When running the genetic algorithm the parameters are as follows in table 5.2 below. To produce a certain amount of fit individuals, the algorithm was run five times, where the top 20 individuals from each run were stored for evaluation at a later time.

Parameters to the Genetic algorithm	
populationLimit	1000
maxGenerations	1000
maxValues	[max value per gene]
minValues	[minvalue per gene]
tmParticipants	3
variation	5
mutationChance	0.4
fitnessGoal	1000
toKeep	2

Table 5.2: The input parameters to the genetic algorithm.

When running the algorithm, the evolution of the average fitness and the most fit individual for one of the runs can be seen below in figure 5.3.

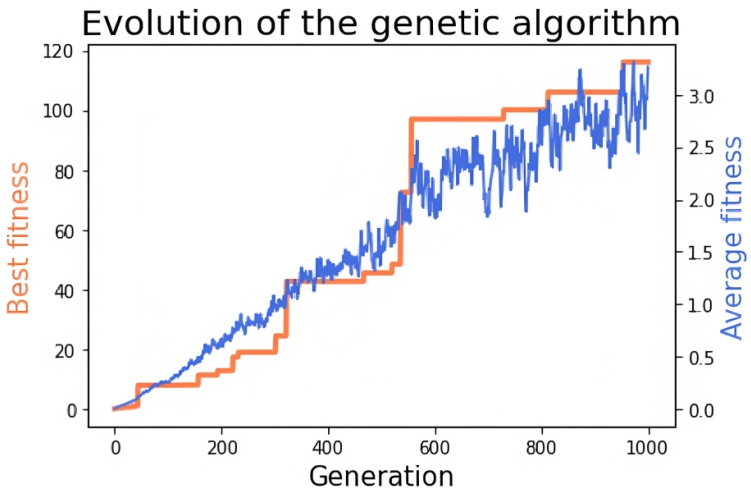

Figure 5.3: Evolution of the fitness, for the most fit individual and the population average. Note that the y-axis for the two lines are different.

As seen in the figure, the average fitness is gradually increasing. The most fit individual is also increasing in fitness, and it is almost possible to see in which generation that individual was created. This could imply that the algorithm is not finding a new most fit individual for each new generation

it is creating and that the *toKeep* parameter is ensuring that the most fit individual is kept in the population.

For evaluation, the combined top 20 individuals from all the runs, 100 in total, were taken into account. Below in figure 5.4 one can see the score, TTC, DTO and speed a run from the algorithm.

	Score	TTC1	TTC2	TTC3	TTC4	DTO1	DTO2	DTO3	DTO4
0	253.928534	3.474589	3.758251	4.489721	2.940320	13.727140	14.935259	12.957363	8.197741
1	213.195171	3.393582	3.675584	4.398112	2.980625	13.797174	15.099935	12.856779	8.364889
2	61.950239	3.393582	3.716762	4.427753	2.991746	13.676496	14.948231	12.971592	8.320352
3	52.875851	3.393582	3.758251	4.478717	2.991746	13.676496	14.970556	12.981991	8.238959
4	42.059893	3.425278	3.705743	4.443334	2.940320	13.848296	15.077840	12.908517	8.257667

	JERK1	JERK2	JERK3	JERK4	Speed1	Speed2	Speed3	Speed4
0	46.883010	81.187119	82.238978	2.181160	4.311790	4.293198	2.857716	2.928372
1	75.417879	10.822945	15.790939	49.278162	4.481156	4.472797	2.912844	2.946740
2	86.656885	42.178431	3.189464	33.464382	4.418493	4.359155	2.925554	2.910097
3	84.726588	73.040675	81.277478	28.806834	4.418493	4.293198	2.867781	2.871709
4	15.066476	12.124975	44.004849	4.914940	4.481156	4.416015	2.893789	2.957621

Figure 5.4: How the top 5 individuals from the genetic algorithm can look like. The features about angular velocity have been removed in the figure to make it more readable.

When comparing the generated individuals to actual collision data, the matches for each run is seen in table 5.3. Here the score is calculated by the formula in equation 5.1. To calculate the score, just the features about TTC, DTO and speed is sent through the equation. The reason for this is that the fitness function only looks at these genes when finding fit individuals, and thus the only "good" genes are within these feature. The *max(Genes)* and *min(Genes)* values are the maximum and minimum value for each feature from the actual collision data. The reason for using this is to measure how far away the row feature and the individual feature is to each other in respect to the value range of the particular feature. A lower is then better then a higher one.

$$score = \sum_{f=0}^{nFeatures} \frac{abs(row_f - ind_f)}{max(Genes) - min(Genes)} \cdot 100 \tag{5.1}$$

In the table, the average score is the average of each row that is found to have a match with a row that is a collision (COL = 1), and the opposite on the right side of the table (COL = 0). To find the matches, the most fit individual gets to find a match first, then that row is excluded from the available rows that the next individual can choose from.

	COL = 1		COL = 0	
Run	Amount	Avg score	Amount	Avg score
1	6	60.8	14	59.5
2	9	54.8	11	60.3
3	9	59.2	11	56.0
4	8	52.6	12	58.9
5	10	53.3	10	52.1
Total	42	56.1	58	57.4

Table 5.3: The matches from the genetic algorithm with actual collision data.

When running the 100 mentioned through the different classifiers of the digital twin, the results are as follows in table 5.4 below:

MLPC	RFC	SVMC	XGBC
56/100	55/100	0/100	19/100

Table 5.4: Predictions that are a collision from the 100 best individuals from five different runs of the genetic algorithm, top 20 from each.

These results are not the very best, especially when looking at the SVM and XGB classifiers. The first one did not believe that a single individual could lead to a collision, whereas the other one only thought 19 if them could. The MLP and RF classifiers thought more of them were a collision, but only a little over 50%.

By looking at the combined results, the individuals generated are maybe not the most realistic ones. There could be multiple reasons for this. One

of them could be the lack of data to compare against to find a good match. Another one can be that the different parts of the algorithm are not good or tweaked enough to be able to find good results. The most likely reason can be that the fitness function inside the algorithm simply is not effective enough to evaluate how good an individual is. Other parts of the algorithm could also be an issue, but they are not as likely as the fitness evaluation, as that is the part which tells if an individual is good or not.

5.3 Answering the research questions

Research question 1

This question is about how machine learning can be used to make a digital twin that can predict collision and avoid them. A digital twin has been created during the work on this thesis and it can indeed avoid a collision by braking. It might not do this under any circumstances, for instance if a vehicle pulls out in front of the vehicle and the distance is too short to be able to slow down.

Research question 2

About RQ2, data was generated with a genetic search algorithm which tried to search for data that could lead to collisions. By following what is mentioned in section 4.6 this was done, but the fitness function can have an update to make the results better.

Research question 3

The last research question is about finding out if a genetic search algorithm can be used to evaluate the digital twin regarding the collision prediction. To do so, the data that is created has to be realistic and that it can represent a situation where a collision is imminent. This was done to a certain extent, as a little under half of the produced individuals matched with a collision

row, as seen in table 5.3. That being said, if the generated data would have matched better to actual collision data and made sure that it is realistic, the data could potentially be used to further evaluate the digital twin's collision predicting capabilities.

Chapter 6

Conclusion

In this thesis there has been developed multiple digital twins for a vehicle. Here the physical twin is the ego vehicle in the SVL Simulator and the digital twins are the multiple classifiers that has been created.

From the results, the classifiers that are created with the Deep Scenario dataset usually detects that a collisions is about to happen, but it also detect it too late, i.e., the vehicle starts to brake, but there is not enough distance and time to avoid a collisions. There could be multiple reasons for that, but the main reason is most likely that the collisions in the dataset have just one value per collision, where the models trained with the generated dataset have multiple collision values before the actual collision. Because of this, those models do not collide, but they are a bit too cautious.

Regarding the genetic algorithm, the results show that the generated data can be used to create testing data, but the data itself might not be the best. When matching the data with actual collision data, a little under half of the data found the best match with actual collision data. When finding a match, the most fit individual gets to find a match from the data first. The matched row is then excluded for the remaining individuals.

When sending the generated data to the classifiers (or the digital twins), the MLP and RF classifiers predicted that half of them were a collision, whereas the SVM predicted that none of them were a collision and the

XGB classifier only predicted 19 out of the 100 individuals.

Future work

To further enhance the capabilities of the digital twins developed in this thesis, there are several potential areas for improvement. One key aspect is the refinement of the machine learning models used in the classifiers, aiming for improved collision prediction. Based on the results obtained, it is evident that the models trained with the Deep Scenario dataset often detect an impending collision but do so too late to prevent it. This issue might be attributed to the nature of the dataset itself, which primarily consists of single collision values, whereas the generated dataset utilized in training contains multiple collision values preceding the actual collision. Consequently, the trained models tend to be overly cautious. By exploring alternative approaches, such as incorporating more diverse collision scenarios or modifying the training process, it is possible to enhance the different model's ability to accurately predict collisions.

Another avenue for future work involves the improvement of the genetic algorithm utilized in generating testing data. Although the algorithm demonstrated the potential to create testing data, the quality of the data itself could be enhanced. One way would be to make the improve the fitness function. The function could maybe be generated by a machine learning model which can understand the different parts of the function better than a human, and therefore produce a better fitness function. Furthermore, exploring adjustments and enhancements to the genetic algorithm itself can lead to the creation of more effective testing data, which can be reliably classified as collisions.

By focusing on these areas of improvement, the machine learning models can be enhanced to provide more accurate collision predictions, and the genetic algorithm can be optimized to generate testing data that better aligns with actual collision scenarios. These advancements will contribute to the ongoing development and refinement of the digital twins, enabling more robust simulations and analysis of collision scenarios.

Bibliography

B. Danette Allen. Digital twins and living models at nasa, 2021. URL https://ntrs.nasa.gov/citations/20210023699. [Visitied: 22.05.23].

Mohsen Attaran and Bilge Gokhan Celik. Digital twin: Benefits, use cases, challenges, and opportunities. *doi: https://doi.org/10.1016/j.dajour.2023.100165*, 2023.

Autocrypt. The state of level 3 autonomous driving in 2023: Ready for the mass market?, 2023. URL https://autocrypt.io/the-state-of-level-3-autonomous-driving-in-2023/. [Visitied: 13.06.23].

Jason Brownlee. What is a confusion matrix in machine learning, 2016. URL https://machinelearningmastery.com/confusion-matrix-machine-learning/. [Visitied: 05.06.23].

Jason Brownlee. How to use roc curves and precision-recall curves for classification in python, 2018. URL https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/. [Visitied: 05.06.23].

I. De Falco, a. Bella Cioppa, and E. Tarantino. Mutation-based genetic algorithm: performance evaluation. *Applied Soft Computing*, 1(4):285–299, 2002. doi: https://doi.org/10.1016/S1568-4946(02)00021-2.

David Eager, Ann-Marie Pendrill, and Nina Reistad. Beyond velocity and acceleration: jerk, snap and higher derivatives. volume 37, page 065008. IOP Publishing, 2016. doi: https://dx.doi.org/10.1088/0143-0807/37/6/065008.

h20.ai. What is multilayer perceptron?, n.d. URL https://h2o.ai/wiki/multilayer-perceptron/. [Visitied: 05.06.23].

Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001. ISSN 0950-5849. doi: https://doi.org/10.1016/S0950-5849(01)00189-6.

Javatpoint. Random forest algorithm, n.d. URL `https://www.javatpoint.com/machine-learning-random-forest-algorithm`. [Visitied: 05.06.23].

Haneet Kour, Parul Sharma, and Pawanesh Abrol. Analysis of fitness function in genetic algorithms. *Journal of Scientific and Technical Advancement*, 1(3):87–89, 2015. ISSN 2454-1532.

LG Electronics America R&D. Python api guide, 2020. URL `https://www.svlsimulator.com/docs/python-api/python-api/`. [Visitied: 22.05.23].

LG Electronics America R&D. Svl simulator: An autonomous vehicle simulator, 2021. URL `https://github.com/lgsvl/simulator`. [Visitied: 22.05.23].

LG Electronics America R&D. Svl simulator sunset, 2022. URL `https://www.svlsimulator.com/news/2022-01-20-svl-simulator-sunset/`. [Visitied: 22.05.23].

Chengjie Lu, Tao Yue, and Shaukat Ali. DeepScenario: An Open Driving Scenario Dataset for Autonomous Driving System Testing, 2023. URL `https://doi.org/10.5281/zenodo.7714194`.

Vijini Mallawaarachchi. How to define a fitness function in a genetic algorithm?, 2017. URL `https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4`. [Visitied: 24.05.23].

Open Source Self Driving Car Initiative. Ossdc sim: An autonomous vehicle simulator (forked from lgsvl simulator), 2022. URL `https://github.com/OSSDC/OSSDC-SIM`. [Visitied: 22.05.23].

The pandas development team. pandas-dev/pandas: Pandas. May 2023. doi: 10.5281/zenodo.7979740. URL `https://doi.org/10.5281/zenodo.7979740`. Version: 2.0.2.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas,

A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Koo Ping Shung. Accuracy, precision, recall or f1?, 2018. URL `https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9`. [Visitied: 05.06.23].

Thomas Plank. Digital twins: the 4 types ad their characteristics, 2019. URL `https://www.tributech.io/blog/the-4-types-of-digital-twins`. [Visitied: 22.05.23].

Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. Lgsvl simulator: A high fidelity simulator for autonomous driving. *arXiv preprint arXiv:2005.03778*, 2020.

Scikit-learn. Support vector machine, n.d. URL `https://scikit-learn.org/stable/modules/svm.html`. [Visitied: 05.06.23].

Goerge Seif. A beginner's guide to xgboost, 2019. URL `https://towardsdatascience.com/a-beginners-guide-to-xgboost-87f5d4c30ed7`. [Visitied: 05.06.23].

Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. Comparative review of selection techniques in genetic algorithm. *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*, pages 515–519, 2015. doi: 10.1109/ABLAZE.2015.7154916.

Chathurangi Shyalika. Population initialization in genetic algorithms, 2019. URL `https://medium.datadriveninvestor.com/population-initialization-in-genetic-algorithms-ddb037da6773`. [Visitied: 24.05.23].

Mooi Lim Siew, Abu Bakar Md. Sultan, Md. Nasir Sulaiman, Aida Mustapha, and K. Y Leong. Crossover and mutation operators of genetic algorithms. *International Journal of Machine Learning and Computing*, 7(1):9–12, 2017. ISSN 2010-3700.

S. N. Sivanandam and S. N. Deepa. Genetic algorithms. In *Introduction to Genetic Algorithms*, pages 15–37. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-73190-0. doi: https://doi.org/10.1007/978-3-540-73190-0_2.

Anna Syberfeldt and Lars Persson. Using heuristic search for initiating the genetic population in simulation-based optimization of vehicle routing problems, 2009. URL https://his.diva-portal.org/smash/get/diva2:227235/FULLTEXT01.pdf.

Synopsis. The 6 levels of vehicle autonomy explained, n.d. URL https://www.synopsys.com/automotive/autonomous-driving-levels.html#c. [Visitied: 13.06.23].

A. J. Umbarkar and P. D. Sheth. Crossover operators in genetic algorithms: A review. *ICTACT Journal on Soft Computing*, 6(1):1083–1092, 2015. doi: 10.21917/ijsc.2015.0150.

Unity. Volvo cars: A unity case study, n.d. URL https://unity.com/case-study/volvo#creating-optimal-vehicle-user-experience. [Visitied: 22.05.23].

John Uri. 50 years ago: "houston, we've had a problem", 2020. URL https://www.nasa.gov/feature/50-years-ago-houston-we-ve-had-a-problem. [Visitied: 22.05.23].

Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

Appendix A

Source Code

The source code for the development can be found on GitHub at:

https://github.com/SigurdGH/MasterThesis