EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Faculty of Science

Autonomous Vision

Practical ML Project

Infraction Visualization and Clustering for better Agent Evaluation in CARLA

Eberhard Karls University Tübingen
Faculty of Science
Autonomous Vision
Alexander Braun, al.braun@student.uni-tuebingen.de,
Luis Winckelmann, luis.winckelmann@student.uni-tuebingen.de
2022

processing period:	17th April - 1st September
supervisor/appraiser:	Prof. Dr. Andreas Geiger, University Tübingen Kashyap Chitta, University Tübingen

Abstract

Virtual driving environments like the CARLA simulator have enhanced and simplified the training and evaluation of self-driving agents drastically. Trained agents can be evaluated safely and efficiently by testing them on different scenarios and recording their performance. However, going through those recordings to find certain failure cases is a time-consuming process. As part of our Practical ML project; we develop a tool to parse the recorded results of an evaluation run in the CARLA simulator and create short video clips showing situations that the agent could not handle and ended in an infraction. Further, we developed an automated clustering pipeline that can be used to find groups of similar infractions that could belong to the same failure mode. In multiple experiments we used those tools and compared different clustering algorithms, parameters and feature sets and clustered the infractions that happened in the evaluation of 3 different driving agents. We found several meaningful clusters showing certain failure cases that seem to appear in multiple agents, as well as problems that are exclusive to a certain model architecture. We show that the result parser and clustering pipeline are useful tools to automate the evaluation process, and that the clusters we found can be used to overcome problems and improve the performance of self-driving agents.

Contents

1 Introduction

In the development of a Self-Driving system, it is crucial to ensure a detailed and careful evaluation of the agent's performance such that the system can handle various different driving scenarios safely. However, training and testing autonomous vehicles in real traffic is both expensive and dangerous. Therefore, computer simulations are often used to efficiently train and evaluate the agents [AL21].

One example of such an application is the CARLA Simulator[1], an open-source simulator for self-driving research. CARLA provides detailed 3D street scenes containing vehicles, traffic lights and pedestrians that follow rule-based behaviors to create many different scenarios and challenging driving situations. A self-driving agent in CARLA gets inputs from simulated cameras and other sensors and outputs lateral and longitudinal driving controls to navigate the car. In the evaluation phase, a trained agent has to drive along several routes in the CARLA simulator and has to reach a certain destination in a limited amount of time. To track and compare its performance, the software keeps track of the driving behavior and especially all kinds of infractions the agent causes, e.g., colliding with other vehicles and pedestrians or ignoring red lights and stop signs. In the end, a driving score is calculated to keep track of the abilities of different agents and report their performance in a leaderboard[2].

While it is an efficient and cheap way to compare different approaches according to a metric, it is often hard to evaluate in detail which situations are problematic for the agent and what causes it to fail there. Of course, one could watch the ego vehicle navigating through the 3D simulation and wait for an infraction to see what went wrong. However, taking the ego perspective it is often not possible to get a good overview of the entire situation and all other vehicles, pedestrians and obstacles that are involved. To overcome this problem, driving scenarios are often rendered in a birds-eye view (BEV) perspective showing the whole scene and all involved cars and objects from above. This is a good way for humans to get an overview of the situation and can be beneficial to find certain failure cases of the driving agent. However, one evaluation in the CARLA simulator usually consists of about 150 km of driving at an average speed of 10-20 km/h. Recording the entire evaluation and rendering it to a BEV video will result in more than 10 hours of video material for every agent

[1] `https://carla.org`
[2] `https://leaderboard.carla.org/leaderboard`

evaluation. Assuming a well-trained agent, the vast majority of the scenes covered in the video are handled well by the agent and only in very few situations it will fail and cause an infraction. Going through the entire, several hours long recording to find and analyze those rare cases is very inefficient and time-consuming work and if we want to evaluate and compare multiple different approaches and agents this process becomes intractable.

The goal of this project is to tackle this problem and develop a tool that visualizes only the critical situations of an evaluation run in the CARLA simulator. Instead of creating one big recording of the entire run, the tool should generate several small BEV video clips showing all infractions and failure cases that happened during the evaluation. The final software we develop provides an evaluation pipeline that automatically parses all available record files and produces short and structured visualizations to be able to focus only on the relevant parts and possible failure cases of the agent. In a second step of the project, we also extracted certain features describing an infraction scene and developed a generalized clustering pipeline to separate the infractions into groups of similar scenarios. This should help to compare different agents and to find cases where multiple agents fail or certain failure cases that are an exclusive problem for one specific approach.

This report documents the important steps and results of the project and is structured as follows: First, we explain the visualization tool and how to set up CARLA. Next we present the clustering pipeline and some examples of resulting clusters we found. Further we provide results from several experiments in which we tried to find clusters of infractions in the evaluation recordings of several driving agents. We conclude the report with a short discussion.

2 Setup & Documentation

In this chapter we focus on the the implementation of the result parser script and the clustering pipeline which is the main part of our project. We document our code and how to set up and use it. We start with a short description about how to install and configure the CARLA environment on Windows. If you do not aim to use any of the covered tools on a windows system, this part can be skipped entirely. Next we cover the result parser script that parses all infractions of an evaluation run and creates short video clips as well as other useful outputs to efficiently investigate the driving behavior of an agent. Here we give a very brief tutorial on how to use the parser followed by a more detailed description of the workflow, inputs and outputs of the script. In a similar way we also talk about the clustering pipeline that we implemented for grouping similar infractions together to find general failure cases of an agent.

2.1 Windows CARLA installation and setup

The first task of our project was to install CARLA on our computers and setup the environment accordingly. On a Linux system the process is very easy and straight forward but when working with Windows it requires a few more steps. Therefore, we want to provide a step by step instruction for installing and configuring CARLA on a Windows system. If you are not working on Windows or already have a running CARLA installation, we recommend to skip this section and continue with Chapter 2.2.

To install CARLA and setup your environment to be able to run the result parser please follow the instructions below:

1. Open git bash in the directory you want to put the transfuser repository:
    - git clone https://github.com/autonomousvision/transfuser.git
    - cd transfuser
    - git checkout 2022
    - mkdir carla
2. Download the following .zip files and extract them in the created carla subfolder (after extrakting, the .zip folders can be deleted):

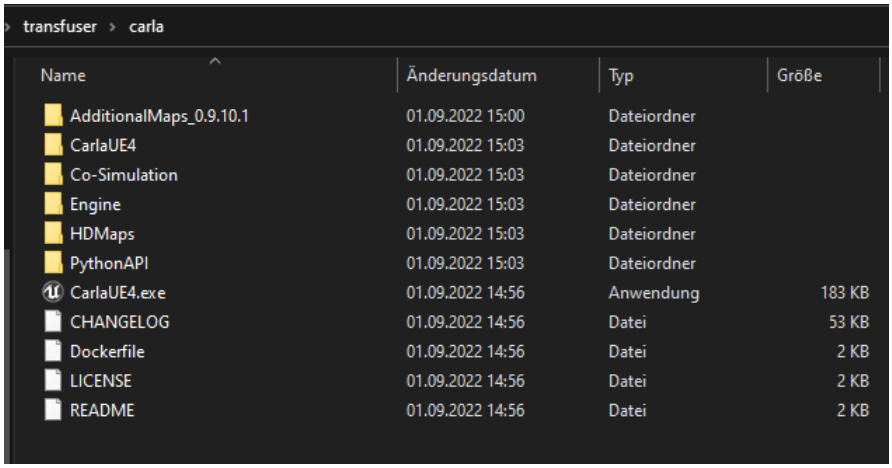


Figure 2.1: Folder structure of the carla installation

- `https://carla-releases.s3.eu-west-3.amazonaws.com/Windows/CARLA_0.9.10.1.zip`
- `https://carla-releases.s3.eu-west-3.amazonaws.com/Windows/AdditionalMaps_0.9.10.1.zip`

3. The content of the CARLA_0.9.10.1 folder should be further extracted giving the folder structure shown in Figure 2.1
4. Open Anaconda Prompt in the transfuser folder and run:
   - conda create –name tfuse python=3.7.13 pip=21.2.2
   - conda activate tfuse
   - pip install –user –upgrade pip
   - cd team_code_transfuser
   - pip install -r requirements.txt
   - pip install setuptools==6.1
   - python -m ensurepip –upgrade
   - cd ..
   - cd carla/PythonAPI/carla/dist
   - python -m easy_install carla-0.9.10-py3.7-win-amd64.egg
   - Install the fitting version of pytorch, e.g.:
     - conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch

- pip install setuptools –upgrade –ignore-installed

5. Confirm the installation by running the parser as taught in Section 2.2

2.2 Result Parser

Before going into a detailed description we give a very short step-by-step tutorial on how to setup and start the result parser script.

1. Setup your system as described in Chapter 2.1 if you are on Windows or else follow the setup instruction of the transfuser repository[0].
2. If you are executing the script for the very first time, start an instance of the CARLA server and keep it running until the result parser is finished. For all further runs of the parser you do not have to open the CARLA server again.
3. **Windows only:** Activate the tfuse conda environment.
4. Open the *parse_results.sh* file that is located under *carla_planning/tools*.
5. Change the *REPO_DIR* parameter to the absolute path to your local version of the *carla_planning* repository.
6. Change the *RESULT_DIR* and *SCENARIO_LOG_DIR* parameters to absolute paths to the directory that contains the result and record files that you want to parse
7. Change the *SAVE_DIR* parameter to the absolute path to the directory to which you want to save the parsed results.
8. **Optional:** If you want to run the clustering pipeline as soon as the result parser is finished, set the *do_cluster_analysis* parameter to true.
9. Save your changes and execute the *parse_results.sh* script.
10. The script will perform the following steps:
    a) Load all result files and extract all infractions
    b) Create the result.csv file
    c) Create overview maps of the infractions in every town
    d) Create a short video clip for every unique infraction
    e) extract features for clustering
11. You can find all created files in the directory that you defined in the *RESULT_DIR* parameter.

Note that these are the minimal steps that are required and for all parameters that can be adjusted, the default values are used. For a detailed explanation please read the following sections of this chapter.

[0] `https://github.com/autonomousvision/transfuser`

2.2.1 Recorded Files and Setup

The main task of the project is the development of a tool that automatically parses the results of an evaluation run of a driving agent in the CARLA simulator. Every evaluation run consists of multiple routes that the driving agent has to pass. For every route two JSON files are created to document the performance of the agent and actions of all involved vehicles and pedestrians. The first file (the result file) contains an overview over all infractions that happened during this route as well as the final score that are calculated for comparing the performance. For each infraction its type and exact location is recorded. In case that there are other vehicles or objects involved, their IDs are also provided. The different infractions are divided into the following types:

- **Collision Pedestrian:** The ego vehicle collided with a pedestrian.
- **Collision Layout:** The ego vehicle collided with parts of the environment, e.g. buildings.
- **Collision Vehicle:** The ego vehicle collided with another vehicle.
- **Outside Route Lanes:** The ego vehicle has left the lane it was supposed to drive on.
- **Red Light:** The ego vehicle crossed a red traffic light.
- **Route Dev:** The ego vehicle deviates from the given route.
- **Route Timeout:** The agent was not able to complete the route within the time limit.
- **Stop Infraction:** The ego vehicle ignored a stop sign.
- **Vehicle Blocked:** The ego vehicle stopped moving. The infraction occurs if the the vehicle does not change its position for a certain amount of frames.

The second file (the record file) contains the position, yaw angle, velocity and size of the ego vehicle (the vehicle that is controlled by the agent) and all other vehicles in its surroundings. Further provided are the positions of all yellow and red traffic lights and the waypoints that describe the planned future position of the ego vehicle. This information can be used to reconstruct the entire evaluation run and render it frame by frame.

To use the result parser tool the following setup is required:

- **Result and Record Files:** The two files which were explained above have to be provided for every route the agent where evaluated on.
- **Route Information:** A xml file is required that contains detailed information about the different routes. For every route a set of waypoints are listed which

the agent has to pass. Also certain weather conditions are simulated for every route which is also saved in the xml file.

- **Town maps:** The routes are distributed over several seperated towns. The outline of each town has to be provided as png files. They are used to mark the location of all infractions on an overview map.
- **Map data:** To render the recorded driving information to a video, the parser has to query the road layout from the CARLA simulator. Therefore a running instance of the CARLA server is required. However to speed up the process the necessary information that is provided from CARLA is automatically saved to a separate directory and the running server instance is only required on the very first time, the tool is started on a machine. For every further run of the parser on that same machine, the server is no longer needed.

2.2.2 Processing Steps and Outputs of the Result Parser Tool

In this section we give an overview of the main steps that are performed when executing the result parser as well as the outputs that are created.

As a first step *all result files are loaded and parsed into a list of infractions*. Since the agent drives on the same route multiple times the exact same infraction could happen multiple times. This is of course important to calculate the leaderboard scores but when generating the video clips we can ignore all duplicates of an infraction. To detect those duplicates we check for every infraction if another infraction of the same type which was processed earlier, happened at the exact same location (i.e. within a very small area around the position of the other infraction). In the end we get a list of all infractions as well as a list containing only the unique infractions.

The next step is *the creation of a file named results.csv*. The file is automatically created and saved to the provided output directory. It can be used to get a first overview over the overall driving performance and the distribution over the different infraction types.

Afterwards the tool *creates overview maps for each town the agent was evaluated on*. As explained above, for each town in the CARLA simulator an image file has to be provided that shows the street outlines of this town. In these image files every infraction that happened during the evaluation is marked on the map as a point at the location of the infraction. The color of the point indicates the type of the infraction that is displayed. For every infraction type a separate town map is created showing only the points according to infractions of this type.

In the next step *the video files for every infraction are generated*. For this we first have to load the record file (see explanation above) for every route to get the position, orientation etc. for all objects in the scene in every frame. In the result.json file we only get the position where the infraction occurred. In order to render a video for

the situation of interest we have to find the corresponding frame in the record file. For this we loop through all states in the record file and keep track of the position of the ego vehicle. If this position is "close enough" to the infraction position we treat this frame as the infraction frame. "Close enough" in this context means that the euclidean distance between the position of the ego vehicle in the current frame to the infraction position is smaller than a certain threshold that can be set in the code of the result parser. The smaller the threshold, the closer we can get to the actual frame of the infraction. But since frames in the record file are only recorded in a certain granularity we also can overstep the correct frame if the threshold is to low. In the end of a successful run of the result parser a list of all infractions is given that were skipped because in no frame the threshold is undershot. If there are many such skipped infraction the threshold should be increased. In general the threshold should be set to the lowest possible number to ensure that the frame that is found in the record file is as close as possible to the actual infraction frame.

If a frame was found to correspond to a certain infraction, we take a fixed amount of frames before and after it. Those frames are the content of the resulting video clip for this infraction. For every frame in the list, all vehicles that are present in the scene are rendered as rectangles to their current location and are rotated based on their yaw angles. For every (unique) infraction a video clip is created in which the infraction frame is approximately in the middle of all frames of the video. The video files are saved to separate directories according to their type of infraction.

In the last step of the result parser we *extract features for every infraction from the record files*. Those features are certain values and properties that should describe an infraction as well as possible. They are later used to cluster them into groups of similar infractions. All features are listed and explained in more detail in Chapter 2.3. The features are saved to a csv file called infraction_features.csv where every line corresponds to one infraction and every column to one feature.

2.2.3 Parameters

In this section obligatory and optional parameters of the parser tool are listed and explained. Many parts of the code contain actions that can be controlled or customized by the user. Therefore we provide quite a long list of parameters that can be set, however most of the parameters are not mandatory and can just be used with their default values if there is no specific need to change them.

- **–xml:** Set the path to find the xml file containing the route information.
- **–results:** Set the path to the directory where all the result files are located.
- **–save_dir:** Set the path to the directory where all the outputs of the parser script should be saved.
- **–town_maps:** Set the path to the directory where the town map images are

located that are used to create the infraction overview maps.

- **–map_dir:** Set the path to the directory where the the town map images in tga format are located.

- **–scenario_log_dir:** Set the path to the directory that contains all record files.

- **–map_data_folder:** Set the path to the directory that contains the map data which is queried from the CARLA server. If the tool is executed for the first time, you most likely don't have this data yet. The tool will provide a message explaining that you have to start the CARLA server and rerun the script to query and save the map data into the directory that you set with this parameter. For all further executions you just have to provide the path to the map data and no running CARLA server is required.

- **–port (Default: 2000):** Set the port to reach the carla server. As described above this is only necessary if the map data is not created yet which is usually the case when executing the script for the very first time. If you did not change the port on which your CARLA server is running you can just use the default value of this parameter.

- **device: (Default: "cuda")** Defines which device should be used for all pytorch methods. Possible options are "cuda"(should be preferred if possible) or "cpu".

- **–no_gif: (Default: True)** If the flag is set, the infraction videos should won't be rendered as *.gif* files.

- **–mp4: (Default: False)** Flag that controls whether the infraction videos should be rendered as *.mp4* files.

2.2.4 Examples for Input and Output

In the following we provide examples for possible inputs and outputs of the result parser and give an example of how to use the script. We assume that we have already evaluated a driving agent in the CARLA environment and have access to the recordings of all routes. For each unique route scenario we have one result.json file and one records.json file (See Chapter 2.2.1). Figure 2.2 shows an example of a result file. In the left image you can see an overview of how the file is structured. In the right image you can see an example of two infractions of the type *collisions_vehicle* that happened in this route.

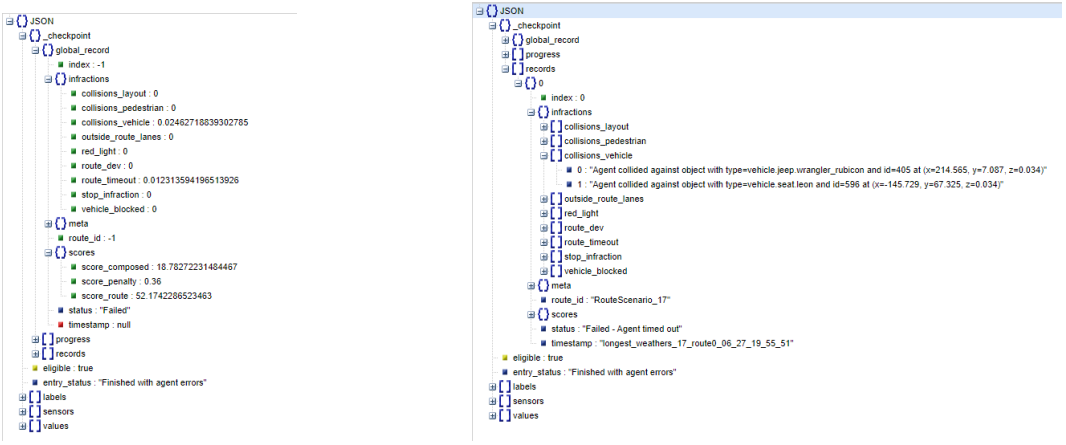


Figure 2.2: Left: Overview of the structure of a result file Right: Example of two collision infractions in the result file

Figure 2.3 shows and example of a record file. In the left image you can see an overview of the structure of this file. In the right image we displayed the values for one recorded state of the evaluation. As you can see, there are three vehicles that would be displayed in this frame because we have three values for positions, yaw angles etc. The first entry in every field always belongs to the ego-vehicle.

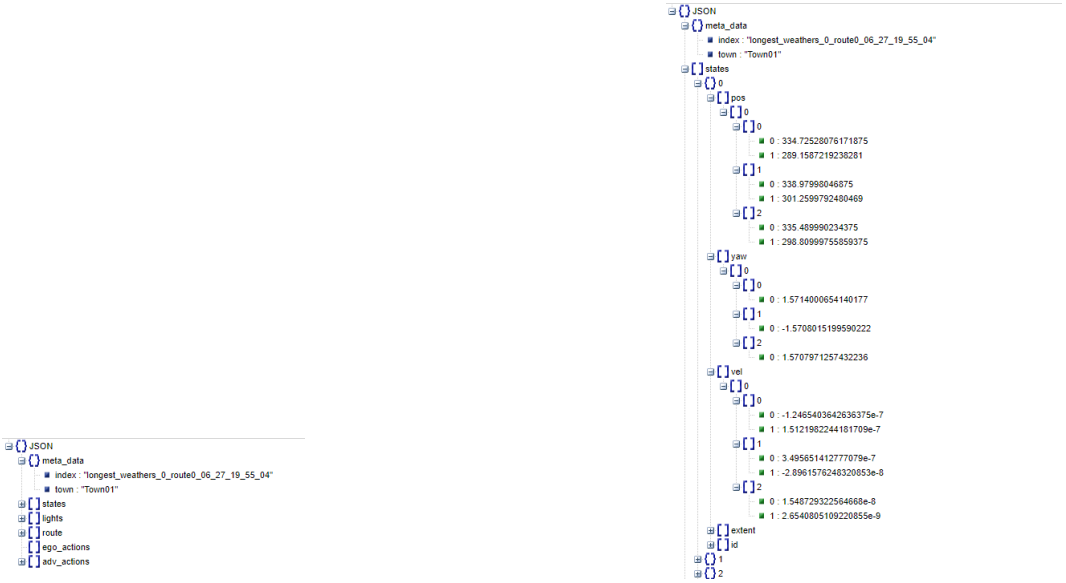


Figure 2.3: Left: Overview of the structure of a record file Right: Values of one recorded state in the record file

All result and record files should be located in the same directory, the result parser

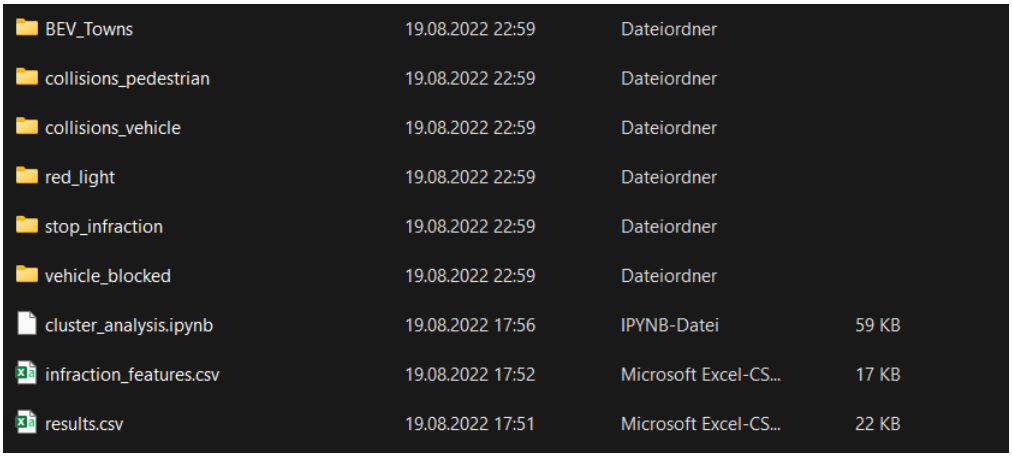


Figure 2.4: Structure of the parser output

will read all of them and match the corresponding files based on the route id. Now we have to make sure that all other required files e.g. the town map images (see Chapter 2.2.1) exist and we know the path where they are saved.

To run the result parser we provide a shell script that contains all parameters as described in Chapter 2.2.3. The parameter values can be adjusted directly in the script. To run the script you need a python installation and the necessary packages. We recommend the conda environment that is provided in the code repository of the *transfuser* project[1]. If you execute the shell script in linux you can set the name of your conda environment in the script such that it will be activated automatically. When using the script on Windows you have to activate the environment manually before starting the script, since this functionality is not supported under Windows.

After the result parser is finished you can find all results in the directory you defined as *save_dir*. The results are structured into sub-directories as you can see in Figure 2.4. Under the root directory you can find the **results.csv**. This csv file contains the overall driving scores that are calculated for the whole evaluation run as well as certain aggregated statistics over different routes and towns. Further it contains a list of all infractions with their type, location as well as weather and daylight condition. Also the number of duplicates of every infraction is provided. which can occur because of the mentioned repetition of each route. Further you can find the **infraction_features.csv** file that contains the extracted infraction features that are used for the clustering pipeline (See Chapter 2.3.2).

The BEV_Towns folder contains the images of the different town maps where every infraction location is marked as a point and its color shows the infraction type. Figure 2.5 shows an example of such an overview map.

[1] `https://github.com/autonomousvision/transfuser`

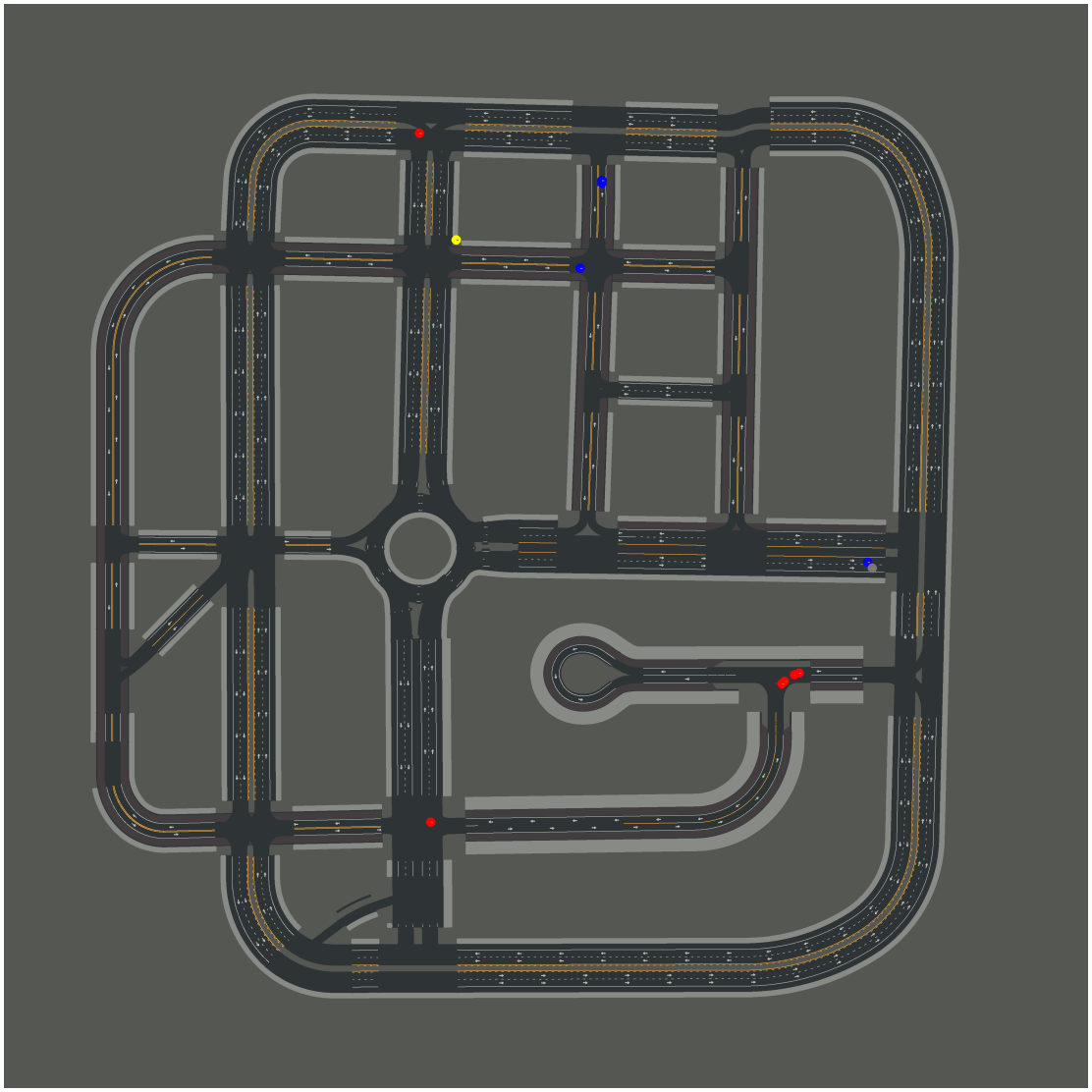


Figure 2.5: Town map that shows the position and type of all infractions that happened in this town. Blue points: collisions_vehicles, Red points: vehicle_blocked, Yellow points: red_light

The other folders contain the rendered video clips separated by infraction type. Figure 2.6 shows an example of the content of such a sub-folder.

The folder BEV_Towns again contains overview maps of every town where the infraction positions are marked but this time only the infractions of the given type are shown in these images. As you can see, there is one gif file for every infraction of a certain infraction type. The file names contain the Id of this infraction, the name of the town and the route id at which this infraction happened. The last segment

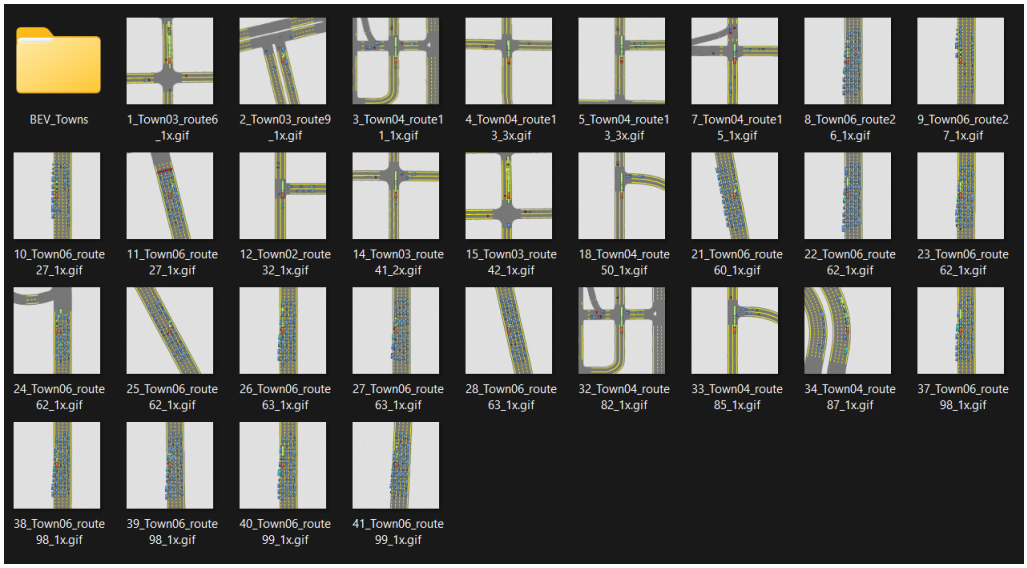


Figure 2.6: Example for generated gif files for infractions of the type collision_vehicle.

in the file name is a counter of how many duplicates this infraction had, where *1x* indicates that this infraction only happened once, *2x* shows, that one duplicate of this infraction was found which was not rendered, and so on. Figure 2.7 shows an example of which objects are contained in the video clips.

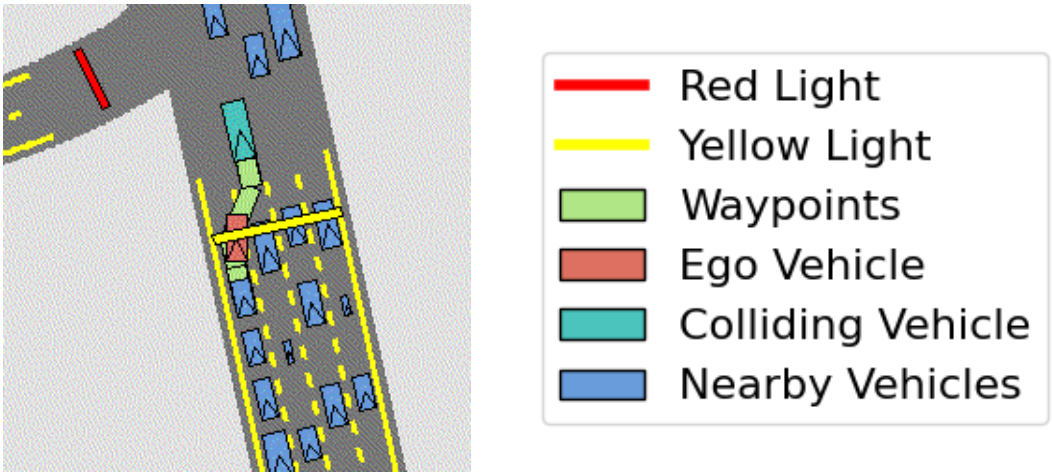


Figure 2.7: Snapshot of a gif showcasing the coloring scheme

The outline of the street is rendered in grey and the lane markings are highlighted in yellow. Further all traffic lights that are either yellow or red in this frame are rendered as yellow or red rectangles. The next few waypoints for the ego vehicle that were predicted by the agent are shown as green rectangles. The vehicles are

displayed as blue rectangles while up to two vehicles are highlighted in different colors. The ego vehicle is always present in the scene and is displayed as a light red rectangle. If the current infraction is a collision with another vehicle this involved neighbor vehicle is displayed in cyan.

In this report we will visualize infractions as a series of 5 key frames that we extracted from the created gifs. Figure 2.8 shows one example of how such a visualization can look like:

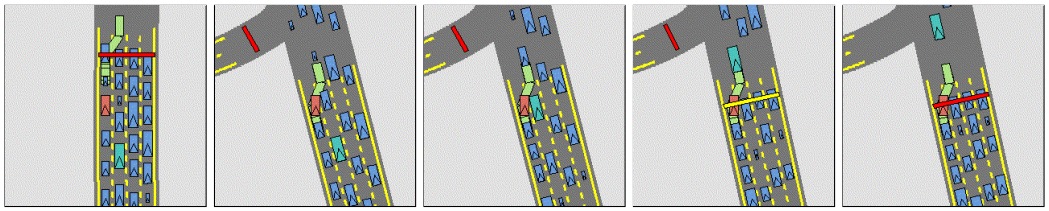


Figure 2.8: Keyframes of a collision infraction

2.3 Clustering Pipeline

2.3.1 Motivation

When we evaluate a driving agent in the CARLA simulator we can observe that it can not handle all situations equally well. When looking at the infractions that are recorded during the evaluation we can see that they belong to different infraction types and occur at different locations in the town. However if we examine those infractions in more detail, especially when watching the birds eye view videos that are rendered by the result parser, we can see that they are not completely different. Instead we can spot several similarities between the failure cases of the agent and we could assign most of them to a few groups of similar infractions. Such a grouping can be very helpful to evaluate the performance of the agent in more detail and beyond numerical metrics. These groups of failure cases can highlight fundamental weaknesses of an agent, that could be tackled in the future. Furthermore the grouping can be used to compare different approaches with each other by checking which groups can be found in the records of multiple agents and which are exclusive to a specific approach.

Grouping the infractions by hand is quite time-consuming and can be influenced by subjective decisions of the human laborer. It would be better to form those groups in an automated and data-driven way. Therefore the second goal of our project was the implementation of an automatic clustering pipeline that takes the extracted features of all infractions of an agent as inputs and clusters them into a variable number of groups.

2.3.2 Features

The properties and the quality of the resulting clustering is highly influenced by the features taken into account. Therefore, one major challenge of the clustering pipeline is to find a subset of all the information that is available in the recordings of the evaluation that describes every infraction in such a way that similar failure cases are close together in feature space while different cases are clearly separated.

While working on the clustering pipeline we experiment with many different feature sets and track the resulting quality of the clusters (more about how we measured this quality in Chapter 3.2). A detailed definition for each of the features as well as the results our experiments with different feature subsets can be found in Chapter 3.5.3

We designed the clustering algorithm to work with an arbitrary number features. The set of features we introduced in Chapter 3.5.3 is just one possible subset that led to good results in our experiments. However, it is absolutely possible and desirable to remove features or add new ones to further investigate the effect on the resulting clusters.

2.3.3 Workflow and Structure of the Clustering Pipeline

In this chapter we want to cover the main functionalities and structure of the implemented clustering pipeline. The goal of this coding task was to develop a script that fully automatically performs a clustering of an arbitrary set of extracted infraction features. The input of the pipeline is a list of feature vectors as they are extracted by the result parser and the output is a list of cluster assignments where each infraction is either assigned to one unique cluster or marked as an outlier.

The results highly depend on the clustering algorithm and the chosen parameters and there is no best choice for every application. Therefore we implemented the pipeline such that it is possible to run the clustering with a whole set of different algorithms and parameter sets. For this we implemented an abstract base class that can be extended to implement an arbitrary clustering method such that every implemented method works with our pipeline as long as it provides the necessary functions.

The clustering pipeline can be imported and used in any other coding project or jupyter notebook. When creating a new instance of the pipeline object you have to provide a path to the extracted infraction features, a list of classes that implements the clustering algorithms that you want to use (the classes have to be extensions of the base class that we mentioned above) and a dictionary that maps each clustering algorithm that should be used with a list of parameters. When executing the pipeline, first the features are loaded from the given path, and then a nested loop runs over all provided algorithms and parameter sets. For every combination the features are clustered and the results are saved in a dictionary. In the end we get a list of clusters

(a nested list where the indices of the infractions that belong to the same cluster are saved in a nested list) for every method-parameter combination.

In certain use-cases we might want to try a large list of different combinations and compare the results to get a feeling for what kind of methods and parameters work well with the given data. It would be intractable to look at every clustering result and compare them by hand. Therefore we added another feature to the pipeline that allows for an automatic evaluation of clustering results. For this you have to provide a dataset of extracted infraction features as well as cluster assignments for those infractions that can be used as ground truth labels. Those assignments can be obtained for example by clustering a few infractions manually. The idea is that additionally to the clusters that were found, the pipeline also returns a certain score that measures the quality of this result (i.e. how close are the predicted clusters to the provided labels) for every combination of method and parameters. Then we could simply sort the results using this score to get a comparison. Since they are different ways to design such a metric we also wanted to keep this functionality as flexible as possible. Therefore we designed the pipeline such that various different metric functions can be used. The pipeline can then be called with a list of such functions and for every cluster results all metrics that were given are calculated and saved together with the predicted clustering in a big dictionary. An example for such a metric and a comparison between different algorithms and parameters can be found in Chapter 3.

Please be aware that depending on the size of the dataset and the number of clustering methods and parameters that are provided, the clustering process may take some time. However since we designed the pipeline to run completely automated and to save all the results, it can just be executed in the background and we can investigate all the results as soon as the process is finished.

Cluster results as lists of indices and single-valued metrics are very useful to automatically compare a large list of algorithms and parameters efficiently. However, in the end, the provided clusters should enhance the evaluation process of a driving agent and help to get a first impression of possible failure cases. For this it is not enough to just output lists of number but we have to visualize the predicted clusters in some way.

The whole clustering pipeline is executed within a shell script (that either can be called manually or automatically from the shell script that executes the result parser). After the clustering process is done and we obtained all results, a jupyter notebook is created and opened automatically. In the notebook we have several opportunities to sort and filter the results. The notebook displays all results and (if labels and metrics were provided) sort them according to the calculated scores. Then we can either define a list of parameters and methods manually or use the combinations that lead to the highest cluster score and visualize the results. For every combination we provided, the found clusters are listed and for every cluster all assigned infraction gifs are loaded and displayed directly in the jupyter notebook. By watching the

infraction clips we can evaluate if the pipeline found meaningful clusters, compare them and get a visual impression of the driving behavior of the agent in certain challenging scenarios and find possible failure cases that happened multiple times during the model evaluation.

3 Results

In the first two chapters of this report we motivated and described the implementation and use of the result parser script and the clustering pipeline. As we already discussed, the goal of these tools is to improve the evaluation process of driving agents in the CARLA simulator, especially improving the search for specific failure cases of an architecture and simplifying the comparison of different approaches. The tools are meant to be part of the general implementation and evaluation process of arbitrary self-driving approaches in the future. However, as an example use case and proof of concept we used the result parser script on actual driving data that was recorded during the evaluation of 3 different driving agents to create video clips and extract features of the infractions that happened during the evaluation. Further, we used our clustering pipeline to find groups of similar infractions and used the created video clips as well as certain metrics that are described in Chapter 3.2 to evaluate the quality of the resulting clusters. We design the clustering pipeline to work with various clustering methods and arbitrary parameter combinations for those methods. For this proof of concept we experiment with several different clustering methods with a wide range of parameter values and compare the clusters that are found. The goal of this experiment is to demonstrate the flexibility and utility of the tools we developed during the project and to show examples of possible infraction clusters.

In the following subsections we further explain the setting and the results of the experiments. First we describe the dataset that we used, i.e. explain what information the data contains and where it comes from. Next we define and explain the metrics that are used in the CARLA simulator to show how the agents are evaluated and compared in the official CARLA leaderboard as well as the metric we choose to measure the quality of clustering results. Further, we briefly describe the model architectures of the driving agents that were evaluated. For this proof of concept we mainly focused on one specific type of infractions, collisions with other vehicles. In the next chapter we explain why we decided to set the focus on this infraction type. Finally, we show the results of the experiments. Here we compare different clustering algorithms and parameter sets and report the combination of method and parameters that led to the most meaningful clusters. For those clusters we visualize different infractions to show what similarities could be found during the clustering.

3.1 Dataset

For the experiments we used a dataset that contains recorded evaluation data from several different existing self-driving agents. For a more detailed description of the different model architectures see Chapter 3.3. We used the data from multiple approaches for several reasons. First this leads to a bigger dataset which gives us the opportunity to analyse more different types of infractions and find more expressive clusters that are well separated from noise and other clusters. Second it is likely that different models behave differently in certain driving situations which leads to different failure cases. Therefore we can compare those approaches and look for failure cases that are exclusive for a certain model and cases clusters of infractions that appear across different agents.

In total the final dataset consists of evaluation data from 3 different agents. Every evaluation in the CARLA simulator consists of 36 separate runs which have different weather conditions, a different time of day and different start and end locations of the route that the agent has to complete. Therefore every run can be seen as a different driving scenario that the agent has to handle [CPJ+22]. Furthermore each of those 36 scenarios is repeated three times for every agent. This is because certain events in the simulator are not deterministic and happen at random locations and random timesteps. Therefore the scenarios are repeated several times to make sure that a certain driving score does not just depend on luck.

So in total this leads to a dataset which contains the recordings of 324 scenarios. For every driving scenario we receive a result.json file as well as a record.json file. Please see Chapter 2.2.1 for a detailed description of the content of those files. The number of infractions that happen in each of those scenarios depends on the conditions (weather, daylight, route) of the scenario and the agent that is evaluated. For our experiments we use the implemented result parser to extract video sequences, and features for every infraction. As we also described earlier, we ignore duplicated infractions, i.e. infractions of the same type that happened at (roughly) the same location and are caused by the same driving agent. The extracted features are then used to find clusters of similar failure cases via the clustering pipeline.

3.2 Metrics

In this chapter we will define and explain different metrics that will be used in the following experiments. We will first cover all important metrics that are used to measure the driving performance of an agent when driving in the CARLA environment. Those scores are used to compare and rank driving agents in the CARLA leaderboard. In the second part we will explain how we measure the quality of the results of our clustering pipeline.

The following metrics are used in CARLA for evaluating agents:

- **Route Completion (RC)**: percentage of the route distance R_i completed by an agent in route i averaged across N routes.

$$\text{RC} = \frac{1}{N}\sum_{i}^{N} R_i$$

- **Infraction Score (IS)**: Aggregation of all infractions summarized via a geometric series. Each infraction j gets a infraction penalty coefficient p_j. The agent starts with a score of 1.0 which gets reduced for every occurring infraction.

$$\text{IS} = \prod_{j}^{\text{Ped, Veh, Stat, Red, Stop}} p_j^{\#\text{infractions of type } j}$$

- **Driving Score (DS)**: This is the metric that the CARLA leaderboard gets ranked by. It is the weighted average of the route completion multiplied with the infraction multiplier P_i,

$$\text{DS} = \frac{1}{N}\sum_{i}^{N} R_i P_i$$

- **Infractions per km**: The total number of infractions normalized by the kilometer driven. Infraction types are mentioned in Section 2.2.1.

$$\text{Infractions per km} = \frac{\sum_i^N \#\text{infractions}_i}{\sum_i^N k_i}$$

  where k_i is the driven distance (in km) for route i

In order to quantitatively compare the results of different clustering algorithms or find a good set of parameters for those algorithms we need some kind of heuristic to evaluate the quality of a set of clusters. In other scientific publications, either a huge dataset and Silhouette Coefficient [Rou87] and Dunn Index [BP95] or a dataset with fixed ground truth is used for quantitative analysis of clustering results. For our use case, the dataset is relatively small and additionally normalizing the dataset let to worse results, which is why we didn't continue with one of those metrics since they only make sense in combination with normalized data. With the hand-clustering labels described in Chapter 3.1, we have a ground truth of sorts, but it is by no means the only way to cluster the infractions, and it is therefore useless to use metrics based on fixed ground truth. A metric that makes sense in our case is the Jaccard index. First introduced in 1912 [Jac12] the Jaccard index can be interpreted as the Intersection over Union (IoU) metric but for sets. Formally the jaccard index between two sets A and B is defined as:

$$\text{J}(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Since A and B in our case are sets of sets, the adjusted Jaccard index must result as follows:
Let A be the clustering results and B be the hand-clustering results, in the form: $A = A_1,\ldots,A_n$ and $B = B_1,\ldots,B_m$ where the A_i and B_j are each a set of indices representing a specific cluster

$$\mathrm{J}_{\mathrm{adj}}(A,B) = \frac{1}{|A|}\sum_{A_i\in A}\max_{B_j\in B}\mathrm{J}(A_i,B_j)$$

Finally, in our use case there is the possibility to declare a cluster as noise. A noise cluster in A or B is completely removed from the corresponding set resulting in less elements in the corresponding set. To implement a noise penalty to induce less overall noise we introduce the noise-adjusted Jaccard index J^{λ}, where $\lambda \in [0,1]$ is the penalty factor.

$$\mathrm{J}^{\lambda}(A,B) = \lambda^{\max(\sum_{A_i\in A}|A_i|-\sum_{B_j\in B}|B_j|,0)}\frac{1}{|A|}\sum_{A_i\in A}\max_{B_j\in B}\mathrm{J}(A_i,B_j)$$

3.3 Models

For our experiments, we selected three different driving agents with access to privileged knowledge about the environment. In this chapter we briefly describe the architectures and the differences between the three models.

- **Entangled MPC:** A standard approach for planning involves search algorithms that look ahead into the future. This approach has achieved remarkable success in classic games such as Go, Chess, Shogi, and Poker [SHS+17, SHM+16, MSB+17]. It has also been extensively applied to robotic control tasks, where it is referred to as Model Predictive Control (MPC) [CB04]. The Entangled MPC approach applies MPC to CARLA via an efficient and accurate proxy simulator of the current driving environment based on a kinematic bicycle model [HRC+22]. This is possible since the agent knows the current position and speed of all traffic members and therefore can roll out their movements into the future to get a good estimation about where they will be located several timesteps in the future. Entangled MPC uses 15 discrete actions that combine 5 steering values for lateral control and 3 throttle values for longitudinal control. The proxy simulator predicts the effects of all 15^4 action sequences that result from taking one of the 15 actions for a horizon of 4 seconds at 1 frame per second. Each action sequence is assigned a scalar cost value based on the distance to other vehicles, proximity to the route, and difference between the predicted and desired target speed. The first action from the sequence with the lowest cost is then executed.
- **Conservative MPC:** To avoid naïvely searching over the entire space of action sequences, which grows exponentially with respect to the MPC time

horizon, Conservative MPC factorizes the action space. Specifically, the actions pertaining to lateral and longitudinal control are disentangled and optimized alternately. Given the current steering value, the model estimates the best throttle, and then uses this throttle value to estimate the best steer. The factorization allows for a more fine-grained simulation rate of 2 frames per second while searching for the best throttle. Specifically, there are 3^8 possible throttle values and 5^4 steer values for a horizon of 4 seconds. In addition to the cost functions of Entangled MPC, this model includes a static safety margin ahead of the vehicle, which penalizes actions that do not maintain a certain amount of free space directly ahead of the ego vehicle based on its current speed.

- **Autopilot:** The autopilot is not a lookahead search model that uses a proxy simulator, but a rule-based agent, taken from [CPJ+22]. The driving policy consists of simple handcrafted rules and applies an A* planner for predicting waypoints as well as two separate PID controllers for lateral and longitudinal controls. To avoid collisions with other traffic participants the model forecasts the future position of all nearby objects by using a kinematic bicycle model. The Autopilot agent is used to provide expert driving data for imitation learning in [CPJ+22]. Compared to the MPC models, the autopilot has several design heuristics. For example, the target speed for the PID controller and the forecasting horizon for collision avoidance are set adaptively based on whether the vehicle is at an intersection or not.

We pick these three models because they show a different driving behavior and therefore we would expect that this also leads to different failure cases during the evaluation. The differences also show up when we look at the driving performance and the number of infractions by infraction type. In Table 3.1 we display the driving scores of the three models as well as the infractions per kilometer for each of the infraction types. Note that we will ignore the stop sign infraction for the rest of this report, the reason for this is explained Chapter 3.4.

Agent	**DS**	**CP**	**CV**	**CL**	**RL**	**OR**	**RD**	**RT**	**AB**
Entangled	51.751	0.0	0.822	0.0	0.060	0.0	0.0	0.394	0.274
Conservative	76.053	0.021	0.260	0.0	0.042	0.0	0.0	0.246	0.091
Autopilot	79.881	0.007	0.236	0.0	0.040	0.0	0.0	0.108	0.088

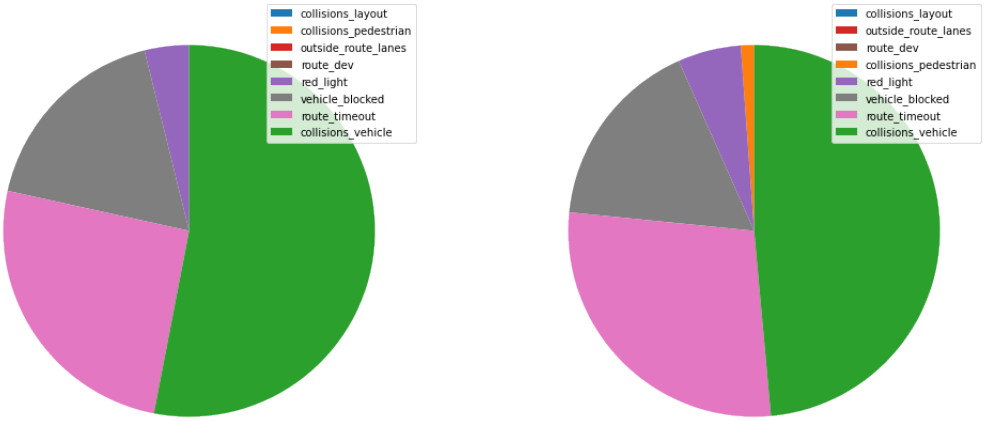
Table 3.1: Evaluation results for the three agents: Entangled MPC, Conservative MPC and Autopilot. The table shows the driving score (DS) and the number of infractions per kilometer by infraction type. From left to right we have: Collisions Pedestrian(CP), Collisions Vehicle(CV), Collisions Layout (CL), Red Light Infractions(RL), Off-road infractions(OR), Route Deviations(RD), Route Time-Out(RT) and Agent Blocked(AB).

From the table it is clear that the Entangled MPC model shows the lowest driving performance and has much more infractions than the other two agents. The Autopilot performs best since this model was carefully engineered with several hand-tuned heuristics. In the following experiments we will further examine and compare the infractions and different failure cases of the three agents.

3.4 Collision Infractions as the Main Focus

As described in Chapter 2.2.1 there are 9 different types of infractions that are recorded by the CARLA simulator. All of them contribute to the infraction score and subsequently to the diving score as explained in Chapter 3.2. To build a robust and save driving agent it is important to tackle all different types of infractions. However, in our experiments we aim to give a clear and compact overview on how to use the result parser and clustering pipeline and how to evaluate and compare different driving agents. Therefore we focus only on one specific infraction type for the rest of this report. Note that all the steps and experiments work completely analogously for every other infraction type and for a full evaluation of an agent we highly recommend to consider all infraction types.

Instead of selecting a random infraction type for this proof of concept we select the type that is the most relevant one for the given driving agents and well suited for showing meaningful clustering results. To find out how relevant each type is, we first look at the distribution of all infraction types over the three model architectures we analysed in our experiments. Please note that we only look at 8 out of the 9 different infraction types. In both model architectures we used for our experiments stop sign infractions are ignored during training and evaluation. Therefore we also decided to not cover this infraction type in our comparison. Furthermore stop sign infractions are not well suited for the clustering experiments since those infractions simply happen if the driving agent fail to recognise a stop sign and they aren't really different groups of failure cases that we hope to find in by using the clustering pipeline. We found out that the remaining 8 different infraction types are very unevenly distributed. For example in Figure 3.1(left) we display the number of infraction by infraction type for the model architecture *entangled_mpc*.



.

Figure 3.1: Left: example infraction-ratio of a model (entangled_mpc). Right: cumulated infraction-ratio over all 3 model architectures that we used in the experiments

It is quite obvious that some infraction types e.g. *collision_layout* or *collisions_pedestrians* didn't happen at all during the evaluation while collisions with other vehicles seem to be the main failure case of this model architecture. In Figure 3.1(right) we show the distribution of different infraction types over all 3 model architectures. Again, we can see that some types of infractions happen far more often than others. Over all model architectures we can observe that the infraction type *collision_vehicle* is the most frequent one.

We wanted to investigate the distribution of infraction types further to make sure that this isn't the case only for the models we are using for the experiments. Therefore we also checked the top-performing architectures on the CARLA leaderboard and analysed how the different infraction types are distributed. Figure 3.2 shows the number of infractions per kilometer by infraction type for the top 6 model architectures on the CARLA leaderboard sorted by driving score [SWC+22, WJC+22, CK22, CPJ+22, CTHM21].

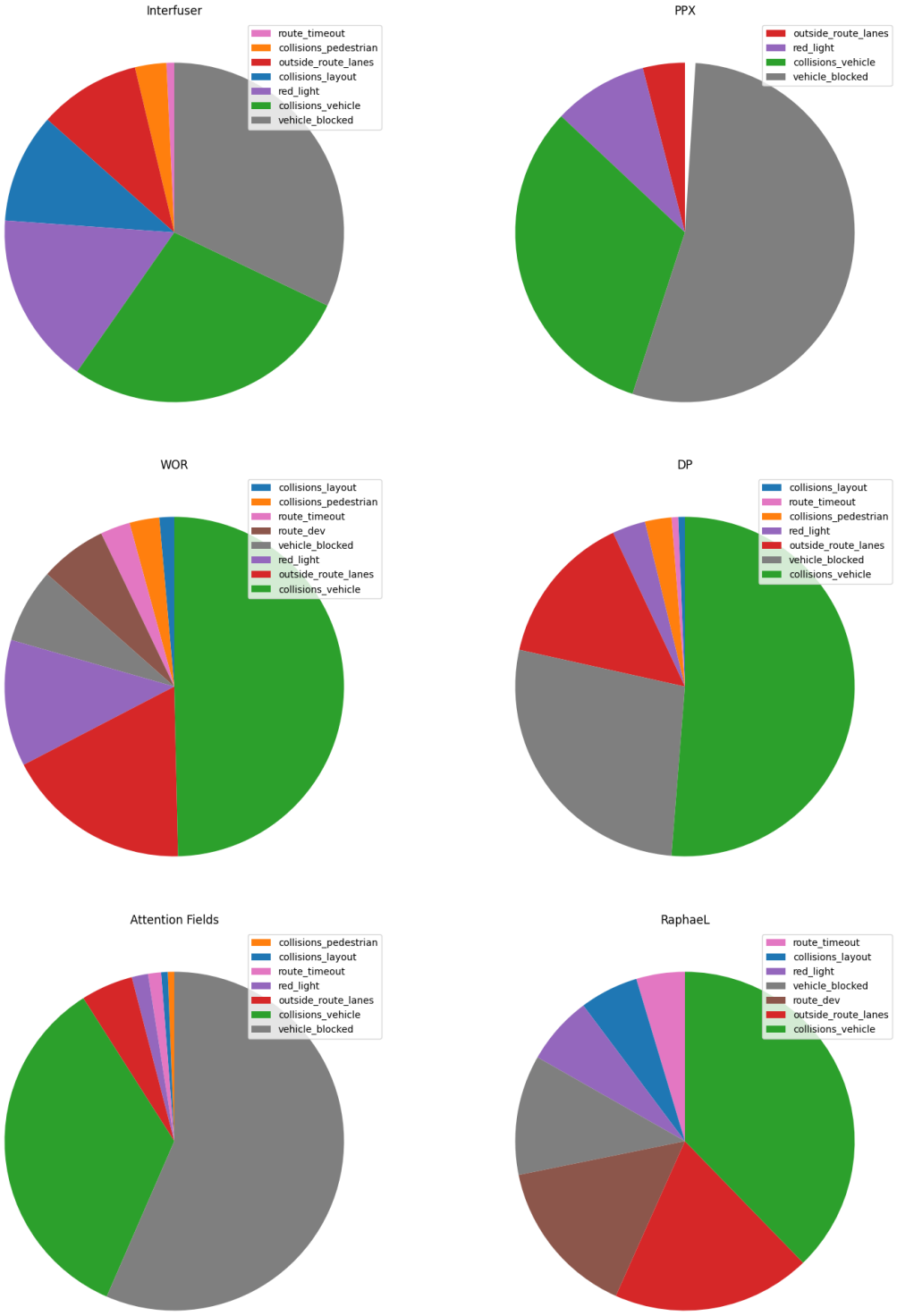


Figure 3.2: Distribution of infractions types in the top performing agents on the CARLA leaderboard

Similar to the distribution of the three models we analysed above we can see that the infraction types are not evenly distributed. Since different model architectures also lead to different driving behaviors it is no surprise that the each model has different failure cases and shows a different distribution over the infraction types. However we can observe that over all 6 architecture the infraction types *collision_vehicle* and *vehicle_blocked* are happening most frequently. For each model *collision_vehicle* is either the most or second-most relevant infraction type which is consistent with our previous results. Based on those statistics and to make our report more concise, we decided to only cover collisions with other vehicles in our experiments. Therefore all further results and visualisations in this report are only considering infractions of the type *collision_vehicle*.

3.5 Experimental Results

In this section we present the final results of our experiments using the parsed results from the 3 different driving agents as a dataset for our clustering pipeline. The experiments are divided into subsections. In the first experiment we used multiple different clustering algorithms and compare them based on the clustering metrics we introduced above. Next we take the algorithm that leads to the best clustering results and have a closer look at its parameters and try to find the best parameter set for this algorithm. We then used the best performing algorithm with optimal parameter set we found and observe its results for different feature subsets to evaluate the impact of certain features on the clustering. In the final experiment we combine all findings from the previous experiments. We use the best performing clustering algorithm together with its best parameter set and the best feature set to find groups of similar infractions. We visualize some of the results to show what kind of cluster were found in the recorded evaluation data. Further we compare the results of different driving agents and find failure cases that are specific to a certain model architecture as well as general failure cases that occur multiple times.

3.5.1 Comparing Different Clustering Algorithms

Dividing samples from a dataset into a number of clusters is a whole research area for itself and there exist many different clustering algorithms, each of them with its specific advantages and disadvantages and different possibilities to control the results by changing certain parameters. There is no "best" clustering algorithm, instead each algorithm may lead to a different result that can provide different insights about the investigated data and may or may not be suited for the current use case. To take this into account we designed the clustering pipeline to work with arbitrary clustering algorithms and any possible set of parameters. For our experiments we picked 5 prominent clustering methods and used them on the extracted infraction features. Before we further describe the experiment we list

and very briefly explain them. Note that we referenced articles that explain each algorithm in more detail.

- **DBSCAN [EKSX96]:** DBSCAN stands for "density-based spatial clustering of applications with noise". This clustering algorithm is suited to find clusters of arbitrary shape and for applications where we expect to have noise, i.e. outliers that don't belong to any cluster. We can set two parameters. The epsilon parameter defines the maximum distance that two points may have from each other to be considered as neighbors. With the second parameter we we can control the minimum number of samples that each cluster must contain. For more details we recommend this article [1] .

- **OPTICS [ABKS99]:** OPTICS stands for "Ordering Points to Identify Cluster Structure". This algorithm is based on DBSCAN and works quite similarly. However, the epsilon parameter as described for DBSCAN is set to infinity by default in order to find clusters of arbitrary density. It is still possible to change this parameter, but it is recommended to do this only to reduce the runtime of the algorithm. The main parameter that can be adjusted is the minimum number of samples in each cluster which can be used to control the cluster size just like in DBSCAN. For further details, we recommend this article [2].

- **KMeans [Mac67]:** The KMeans algorithm is probably the most prominent clustering method. The goal of this algorithm is to find K cluster centroids and then assign every datapoint to the cluster with the closest centroid. By alternatingly assing points to clusters and then updating the cluster centroids to be in the centers of the new formed clusters the algorithm eventually converges to a stable state. The only parameter to set is the number of clusters K. Since we do not know the true number of clusters beforehand, KMeans should be applied mulltiple times with different values of K. A more detailed explanation can be found in this article [3].

- **Hierarchical [Mur83]:** Hierarchical Clustering is an algorithm that finds nested clusters by either iteratively merging or splitting clusters. When using agglomerative clustering every point in the datset starts as its own cluster and the algorithm proceeds by merging clusters which are close to each other. Divisive clustering works the other way around. Here we start with only one cluster containing all points and iteratively splitting it up into smaller groups. As parameters we can set the linkage criterion which specifies how to calculate the distances that are used to determine how to split or merge the clusters. Further we have to either specify a certain number of clusters or a distance threshold to control at which point the algorithm should stop. For more details

[1] https://towardsdatascience.com/dbscan-clustering-explained-97556a2ad556
[2] https://towardsdatascience.com/clustering-using-optics-cac1d10ed7a7
[3] https://towardsdatascience.com/understanding%2Dk%2Dmeans%2Dclustering%2Din%2Dmachine%2Dlearning%2D6a6e67336aa1

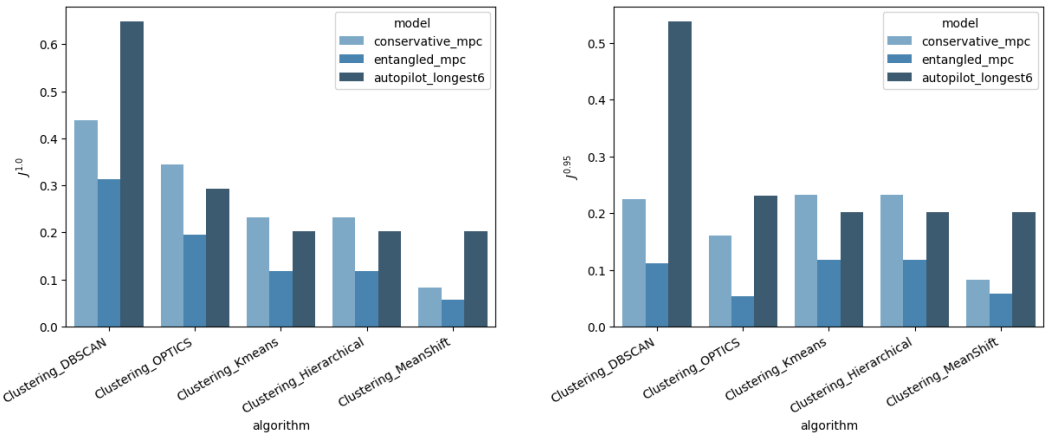


Figure 3.3: Left: Best results of each algorithm for Jaccard with $\lambda = 1$ when doing a large scale grid search. Right: Best results of each algorithm for Jaccard with $\lambda = 0.95$ when doing a large scale grid search

have a look at this article [4].

- **MeanShift [Che95]:** The MeanShift algorithm works in a similar way as the KMeans algorithm. The algorithm updates centroid proposals to find cluster means in regions with a maximum density of points. In contrast to KMeans, the algorithm automatically determines the number of clusters instead of enforcing a fixed number. The only parameter to set is the so called bandwidth that controls the size of the area in which we search for the point with the highest density. To learn more about the MeanShift algorithm we recommend reading this article [5].

We compare the results of these 5 different algorithms by looking at the Jaccard index we introduced in Chapter 3.2 both with and without the noise penalty. Figure 3.3 shows the results of this comparison. We displayed the Jaccard index for every clustering method. Because the results differ depending on the model architecture for which we analyse the infractions, we provide the clustering metrics for each model separately.

In both cases we perform a large scale grid search over different parameter sets for every algorithm and provide the best score for every model architecture respectively. As we can see in the two figures there are significant differences in the Jaccard index over the different algorithms and also over the data of the different driving agents. Both for the Jaccard metric with and without noise penalty and we can observe that the *DBSCAN* algorithm achieved the highest scores. When choosing the

[4] https://towardsdatascience.com/understanding%2Dthe%2Dconcept%2Dof%2Dhierarchical%2Dclustering%2Dtechnique%2Dc6e8243758ec

[5] https://towardsdatascience.com/understanding%2Dmean%2Dshift%2Dclustering%2Dand%2Dimplementation%2Dwith%2Dpython%2D6d5809a2ac40

amount of noise penalty we reach a trade-off that we encountered regularly during the experiments. Some algorithms output very meaningful clusters that are well separated from each other but have a large amount of noise samples which means that many of the samples were ignored and don't show up in any of the clusters. Other algorithms (and also parameter sets) however treat just a few samples as noise but result in overlapping clusters that are not that expressive anymore. Again, the answer on what algorithm and parameter set should be used highly depends on the use case and in practice it may be the best choice to look at different cases because all of them can provide a different viewpoint on the dataset and can reveal different groups of failure cases that all may be relevant for the evaluation of a driving agent.

Based on the results of this experiment we select the *DBSCAN* algorithm for all further experiments since it leads to the most expressive and well separated clusters. Since the main goals of our experiments are to give examples of possible failure cases and to show how to use the clustering algorithms we decided to prefer expressive and non-overlapping clusters over a small amount of samples that are treated as noise and therefore we focus on the Jaccard Index without noise penalty in the further experiments.

Please note that we perform this comparison to get rough evidence on the performance of the different algorithms and the fact that we picked one of them is just to keep the following experiments compact and simple. When using the clustering pipeline to evaluate driving agents in practice we highly recommend to try different clustering algorithms and use the Jaccard Index just as a rough orientation. It might be much more useful to use several algorithms with multiple different parameter sets and use all different results to analyse the failure cases of the model instead of only using the setup that lead to the highest clustering score.

3.5.2 Observing the Impact of Different Parameter Sets

In the previous chapter we introduced a number of different clustering algorithms that we evaluated during our experiments. We also explained the parameters that can be modified to control certain aspects and the results of the algorithm. It is quite obvious that different values and choices for these parameters can highly affect the clusters that are predicted. For the same reason that there is no overall best clustering algorithm, there is also no overall optimal parameter set for those algorithms. Instead different parameter choices may be suited for different applications or they simply can reveal different properties of the dataset. Therefore we highly recommend to try out different parameters and compare the resulting clusters when using the pipeline to evaluate an agent in practice.

However to keep our experiments simple we wanted to pick one algorithm and find one parameter set that outputs meaningful clusters. In Chapter 3.5.1 we already decided to proceed with the DBSCAN algorithm. In this chapter we apply DBSCAN with a large set of different parameter values and try to find the parameter

combination that maximizes the Jaccard score of the resulting clusters similar to the process in Chapter 3.5.1.

In the previous chapter we already listed the main parameters that can be set when using the DBSCAN algorithm. We now want to explain all of them in detail and discuss the final grid we decided on and the reasoning behind it:

- **eps:** The epsilon parameter defines the maximum distance for two samples to be considered as neighbors of each other. This is the most important parameter of the algorithm since it highly affects the decision if two infractions are similar or not. Decreasing the epsilon parameter will lead to clusters that are separated more strictly since samples have to be very close together in feature space to be considered as member of the same cluster. However a small epsilon value also leads to a larger amount of samples that are classified as noise since it is more likely that there are not enough samples in a neighborhood to form a clusters which results in treating them as outliers. Prior work from Satopaa et al. [SAIR11] and Schubert et al. [SSE+17] give intuition and tools to find a range for the epsilon parameter. However, the dataset we are using for clustering lacks the size to make use of them. Instead, we can make use of the small size of our dataset by simply extending our grid for epsilon to a large size and analyzing the values by looking at the Jaccard metric for each parameter constellation. Figure 3.4 and Figure 3.5 show the values for eps sorted by the $J^{1.0}$ metric. We used a fine-grained grid where we ranged the values for eps between 0 and 1500. The plots show that the best results get reached for a eps-value of $4.5 \leq \text{eps} \leq 6.5$ while even when looking at the top 5000 values, the main singled-digit eps values seem to perform the best. For the final analysis we decided on a grid for eps that does steps of size 0.01 from 0 to 100 and steps of size 0.15 from 100 to 1500.

- **min_samples:** With this parameter we can control the minimum size of each cluster. A small value (in particular in combination with a small epsilon value) will lead to a large number of clusters which are very specific. If we aim for fewer clusters containing failure cases that are similar in a more broader sense, we have to increase the value for this parameter. For min_samples the grid choices are simpler. Since it can only be an integer values and we don't want to cluster single points into a cluster a minimum of 2 is required. However, we don't want to enforce very large clusters. Thus, for the rest of our experiments the grid for min_samples ranges from 2 to 5 ($2 \leq \text{min\_samples} \leq 5$).

- **p:** This parameter controls the power of the Minkowski metric to be used to calculate distance between points. If None, then p=2 is used. We decided to include $p = 1$ as well in our grid search.

- **algorithm:** By default this is set to *auto*. In our experiments we found out that setting this parameter to *kd_tree* results in a 5× speedup and thus we decided to stay with the algorithm parameter on kd_tree for all of our results.

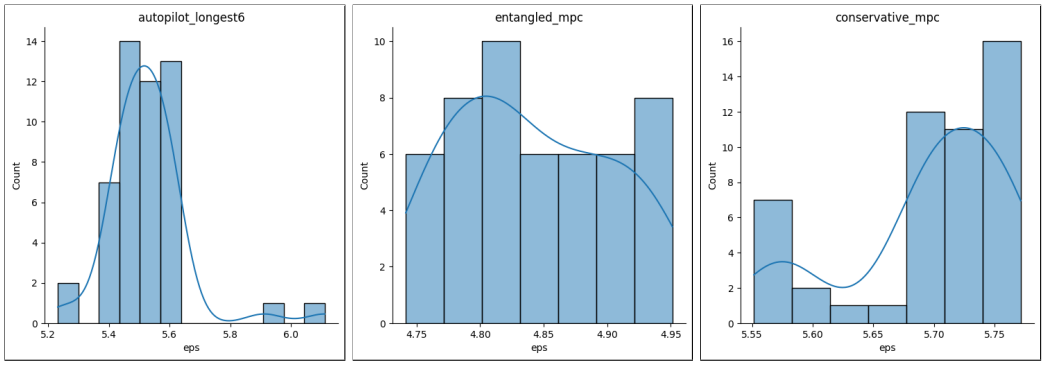


Figure 3.4: Distribution of the top 50 parameter choices for eps.

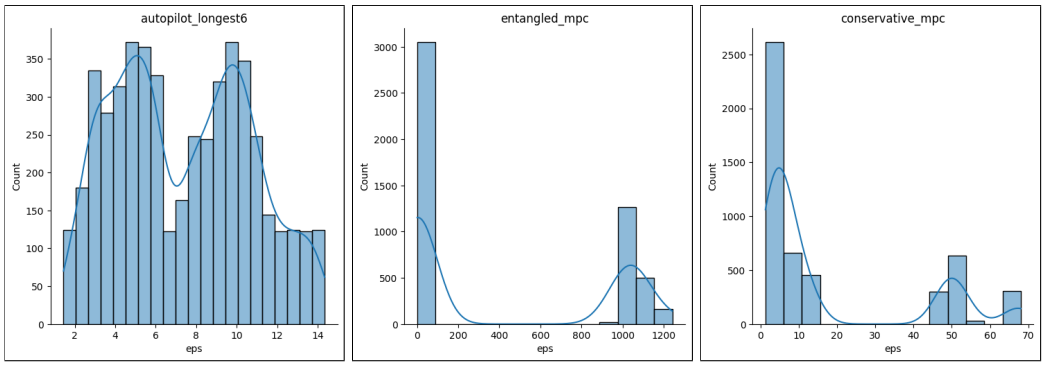


Figure 3.5: Distribution of the top 5000 parameter choices for eps.

In the end we chose the final parameters for each algorithm as shown in Table 3.2.

3.5.3 Comparing Different Feature Subsets

As already described in Chapter 2.3.2, we designed the clustering pipeline to work with an arbitrary subset of categorical and numerical features. In the following we list and explain the different features that are extracted for every infraction when executing the result parser script:

Agent	eps	min_samples	p	algorithm
Entangled	4.7953	3	2	kd_tree
Conservative	5.7362	3	2	kd_tree
Autopilot	5.846	3	1	kd_tree

Table 3.2: The final choices of parameters for each model

- **Yaw angle of the ego vehicle:** This feature expresses the orientation of the ego car.

- **Velocity of the ego vehicle:** The velocity of the ego vehicle is expressed with two separate variables. One showing the velocity in x-direction and the other in y-direction.

- **Waypoints of the ego vehicle:** This information also consists of multiple features, To express the route that the ego vehicle was going to take at infraction time, we fit a polynomial function to the waypoints of the ego car at this point in time. Depending on the order of the polynomial we get a fixed amount of parameters for describing the fitted curve. For the current version we have chosen a polynomial of order 3 which lead to a fitted curved defined by 4 parameters. Those parameters are treated as 4 separate features in the feature space. The order of this polynomial can be easily changed in the script resulting in a more fine-grained fit with more parameters or a more coarse fit with less parameters.

The features listed above are available for every infraction. However if the infraction is a collision with another vehicle, we can add additional features based on the neighbor vehicle that was involved in the collision. Those features are as followed:

- **Relative position of neighbor vehicle:** Here we have two feature describe the difference in x and y direction of the position of the ego vehicle and the position of the neighbor vehicle. This offset describes where the neighbor vehicle was located relative to the ego vehicle at the time step when both cars collide.

- **Yaw angle of the neighbor vehicle:** Analog to the same feature for the ego vehicle we describe how the neighbor vehicle is rotated.

- **Extent of the neighbor car:** The cars in the CARLA simulator have different sizes. We have access to the four corner points of the rectangle describing the bounding box of the neighbor vehicle. To take its size into account we included those corner points in the feature space. Because we have four points with a x and y coordinate each we end up with 8 separate features describing the extent of the neighbor car.

In this experiment we analyze the quality of the cluster results (measured by the Jaccard metric) when using only certain subsets of these features. For this we used the DBSCAN Algorithm with the set of parameters that we found in Chapter 3.5.2 and run it multiple times while in each run we removed one or multiple features from the dataset. We keep track of the Jaccard score of the resulting clusters for each feature subset. In Figure 3.6 we visualize the results of the experiment.

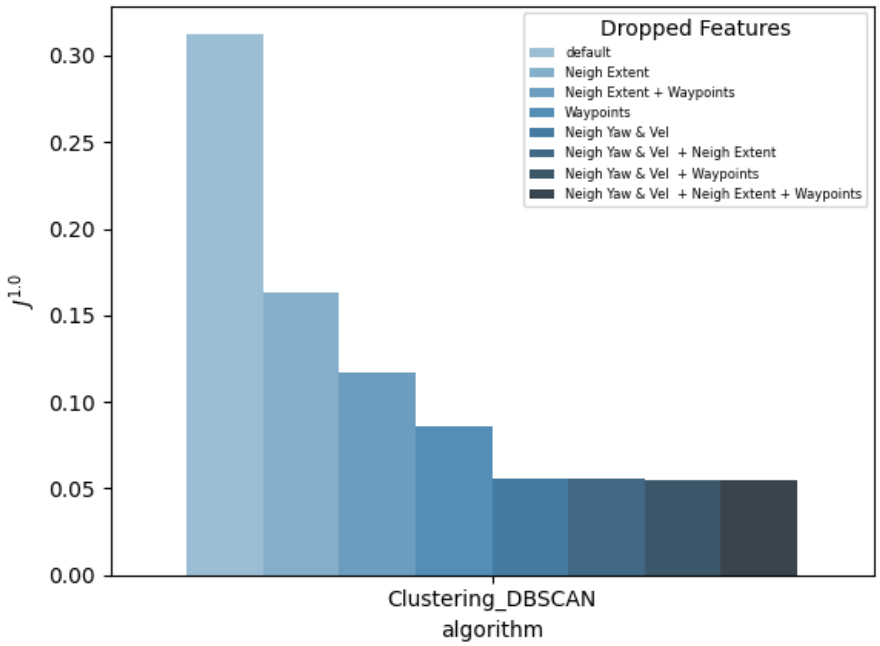


Figure 3.6: Overview of the impact when dropping different features to the results Jaccard with $\lambda = 1$ of the entangled model.

The bar that is described as *default* shows the result when using the complete feature set. Every other bar is described in the legend with the feature or the set of features that was removed. We can observe that the full feature set achieves by far the highest Jaccard score which is gives evidence that all of the features contribute to the resulting clusters in a positive way. However by comparing the performance drop when removing certain features we see that they do not all contribute with the same amount. Dropping the extents of the neighboring vehicle for example did not have a much of an impact as removing its yaw angle or velocity. If we look at some of the clusters we formed manually, this result makes sense since in most of the clusters the main similarity between the different infractions is movement and speed of the vehicles before they collide while their sizes don't matter that much. Therefore, it is reasonable that the yaw and velocity of the neighbor car is more important for the clustering algorithm than its extent.

The main insight of this experiment is that we achieve the highest clustering score when using the full feature set and removing certain features will decrease the performance. Therefore we will use the full set of features in the final experiment.

3.5.4 Evaluating Different Agents by Observing Clusters and Failure Cases

In the previous experiments we compared different clustering algorithms, parameters and feature subsets. In the final experiment we use those results to configure the clustering setup that lead to the best results in terms of the Jaccard metric. We use this setting to cluster all infractions of all different model architectures. In this chapter we visualize the results of this clustering and give examples of groups of similar infractions that describe general failure cases. Further we compare failure cases between the different model architectures to find unique and shared clusters.

First we look at every agent separately and show some examples of clusters that we found using the pipeline. We will display every cluster in a table-like figure where every row corresponds to a unique infraction and the 5 images in every row show 5 key frames that we extracted from the video clips to visualize the infraction. The third element in every row is highlighted with a red border and contains the infraction frame.

We start with the model architecture *Conservative MPC*. In Figure 3.7 you can see a cluster that consists of 5 different infractions. All except of one of the infractions show rear-end collisions where a neighbor car crashes into the ego vehicle from behind. The last example shows a collision where another car merges into the ego vehicle from the right side and might be falsely assigned to this cluster. In 3 of the rear-end collisions the ego vehicle drives straight while in the fourth one the collision happens after a lane change. In all of those cases we could not spot a clearly incorrect behavior of the agent and the collisions seem to be caused by the other vehicle.

In Figure 3.8 we display a cluster of 3 infractions. All scenarios look very similar and show a highway scene where a car merges into the ego vehicle from the left. The maneuver of the neighboring car could be explained by the fact that the very left lane is ending a few meters ahead such that all cars from this lane have to merge into the right lane. Again we could not find any misbehavior of the agent and the collision is most likely caused by the other vehicle.

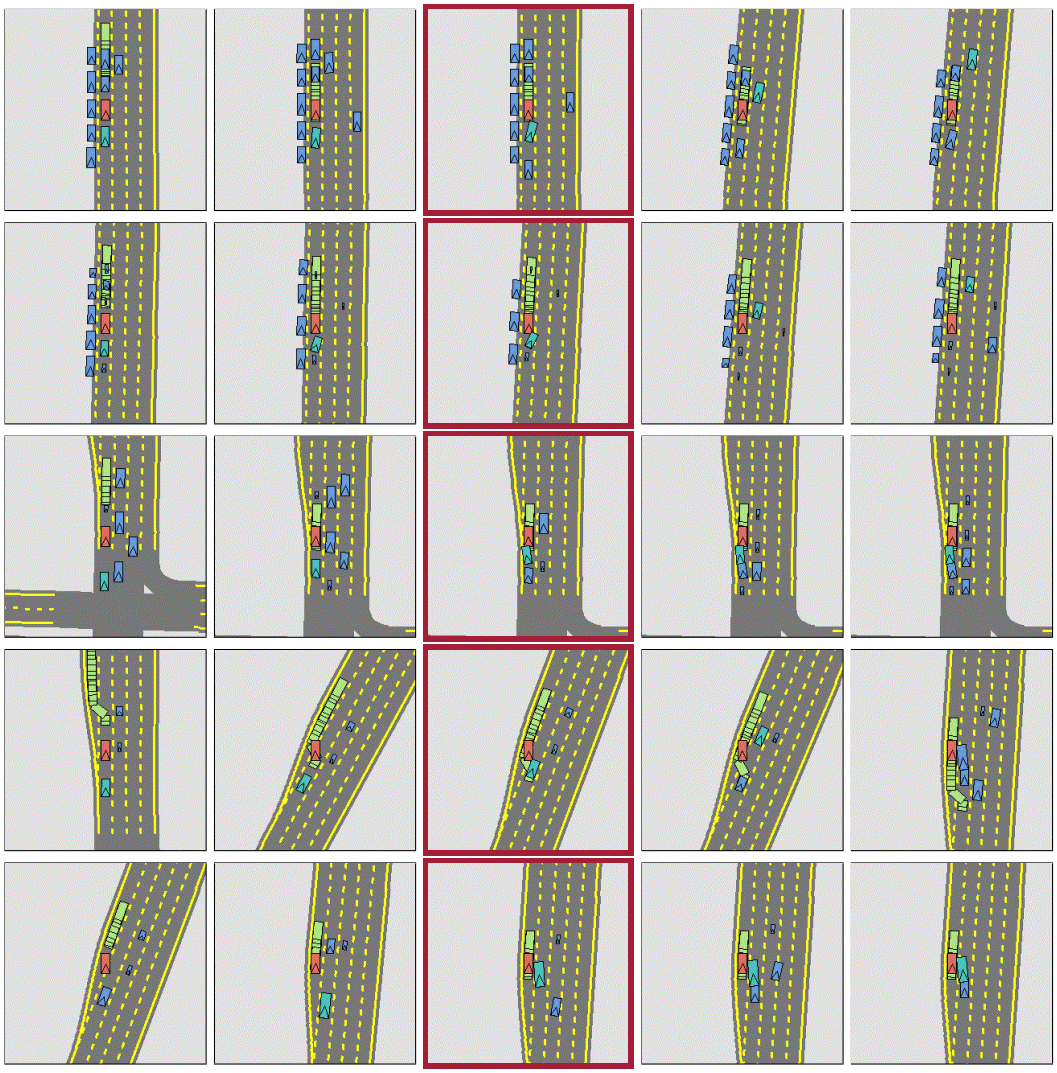


Figure 3.7: Cluster 1 of the conservative_mpc agent. The cluster mainly contains rear-end collisions in a highway scenario.

Figure 3.9 shows another cluster with 5 infractions. Case 1 and 5 both show an intersection where the the ego vehicle drives straight and is hit by another vehicle from the right. The remaining 3 infractions look very similar to the highway scenes in Figure 3.8 that we saw above. Here we have a case where the clustering algorithm had put together two different clusters where the three highway scenes would actually belong to another existing cluster. This is a good example of an output that may change if we would use different parameters. When analysing the scenario in infraction 1 and 5 it is quite probable that the neighbor car would have the right of way while our agent keeps driving and causing the collision.

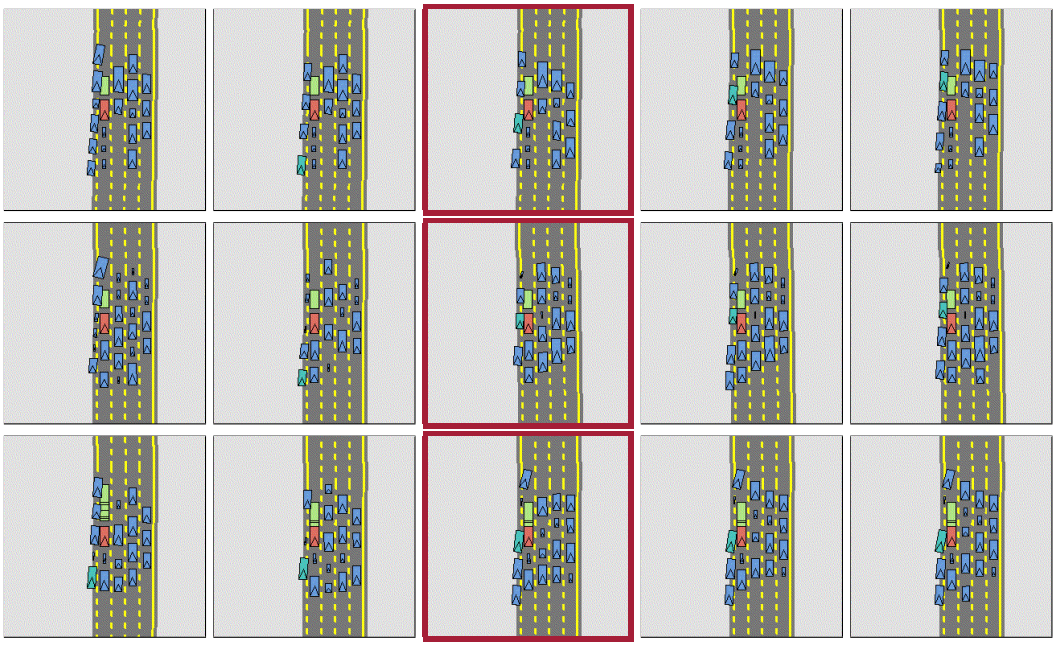


Figure 3.8: Cluster 2 of the conservative_mpc. The very left lane of the highway ends, forcing the vehicles to change the lane. In all three examples we can see that one vehicle merges into the ego vehicle from the left because of this lane change.

Let's continue with the *Entangled MPC* agent. This model has a much lower driving score in comparison to the other two agents and consequently also much more infractions. After visualizing the clusters we notice a certain failure case that happened quite often and which we only found for the *Entangled MPC* agent. This cluster is visualized in Figure 3.10. Actually this cluster contains 18 infractions but we only displayed a few examples. As you can see the clustering pipeline grouped together very similar collisions. In each of them we can see that the ego vehicle tries to turn into the street to its left and has to cross the oncoming traffic lane where it collides with a car that drives straight on this lane. In all cases the agent seems to not recognize or ignore the oncoming car leading to this infraction. This example is particularly interesting since we found a failure case that happens multiple times but exclusively for one specific model architecture.

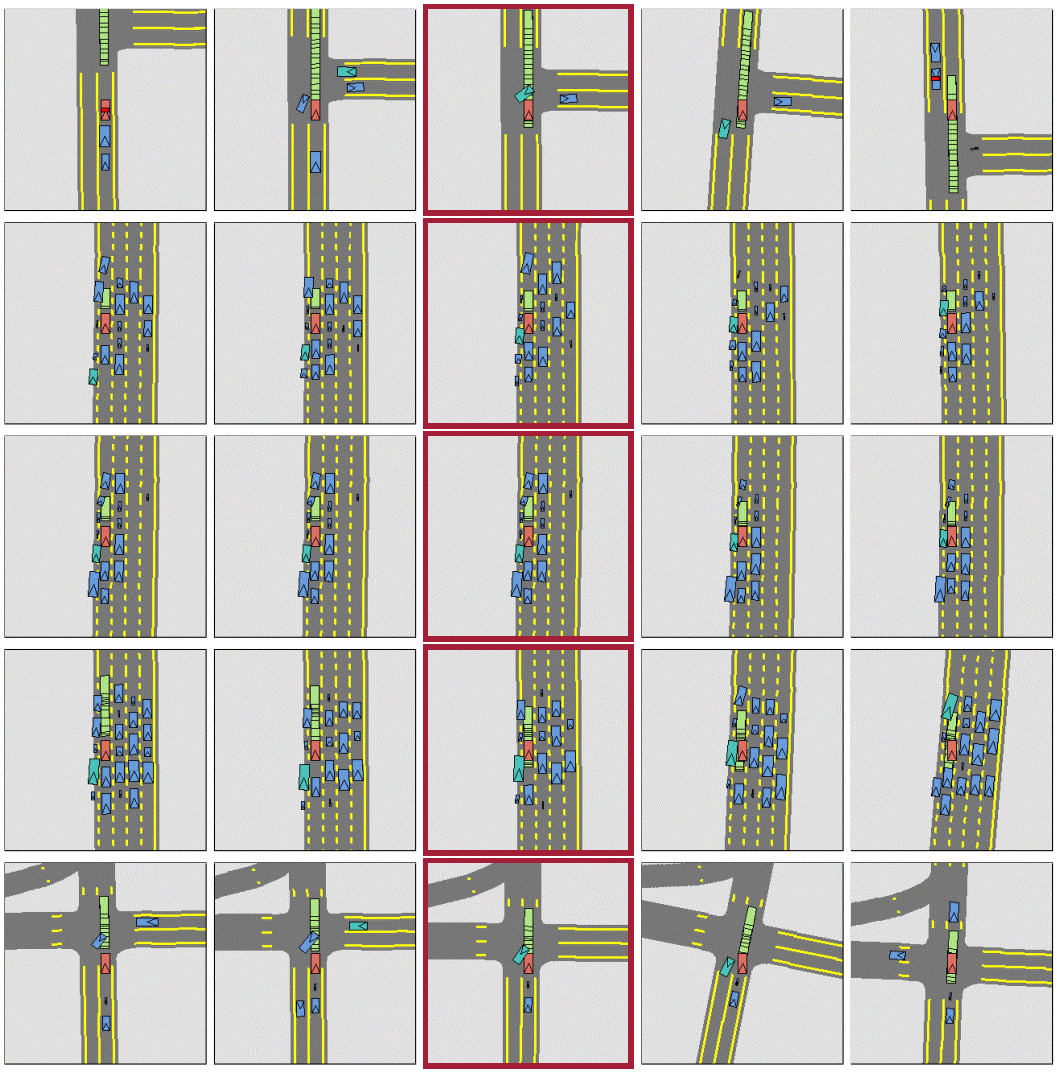


Figure 3.9: Cluster 3 of the conservative_mpc. We can spot two different failure cases. In the first case the ego vehicle drives straight across an intersection and is hit from the right. The second case shows again the lane change scenario that we covered above.

The cluster which is displayed in Figure 3.11 consists of 5 infractions. With the exception of one case, the other examples show a very similar situation to the case that we observed for the *Conservative MPC* agent where the ego vehicle is driving straight across an intersection and is hit by a car or bicycle from the right. Again we may have found a general failure case for this agent where it ignores the right of way of the vehicle from the right and keep driving straight.

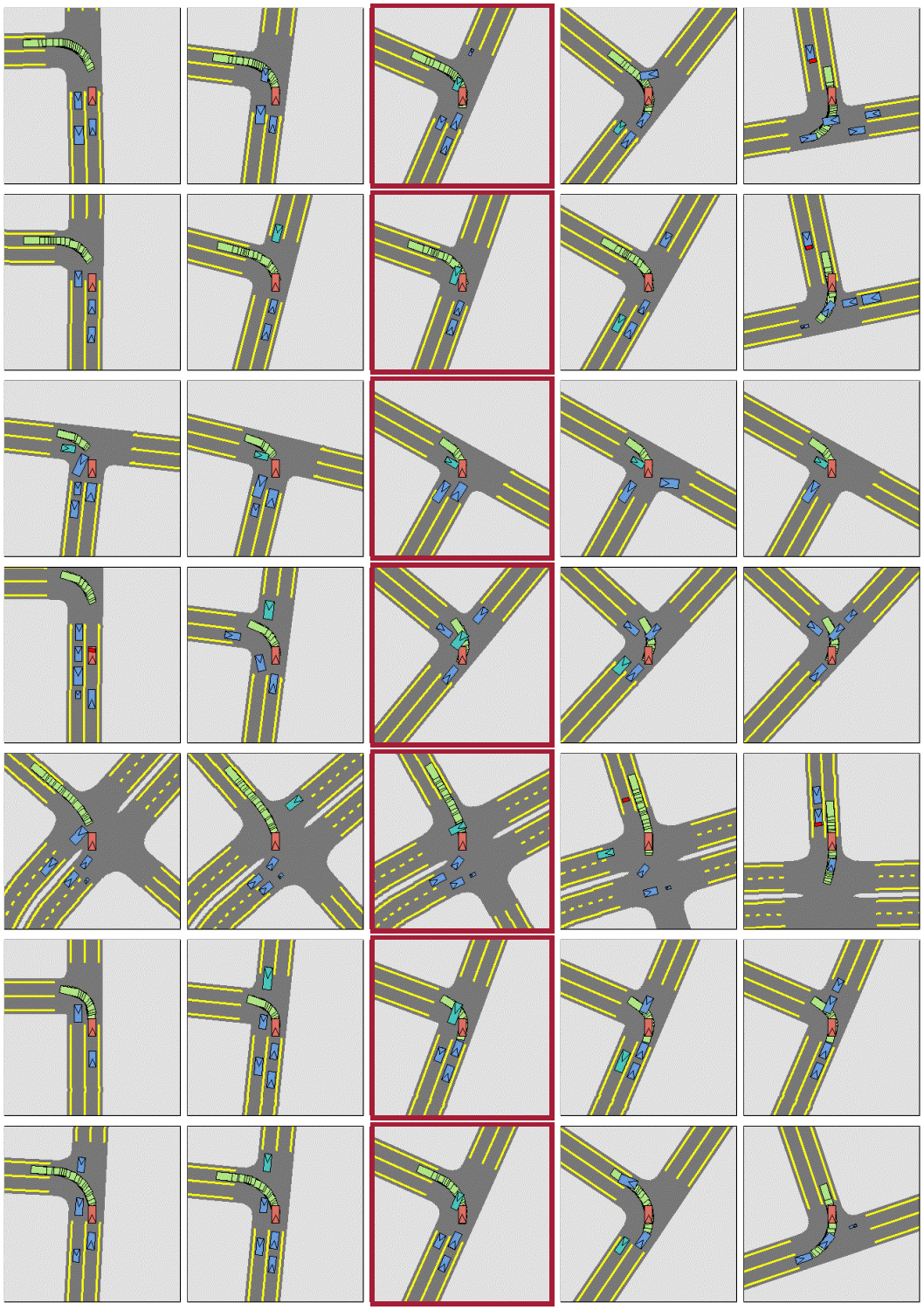


Figure 3.10: Cluster 1 of the entangled_mpc. The ego vehicle tries to do a left turn and collides with a car on the oncoming traffic lane.

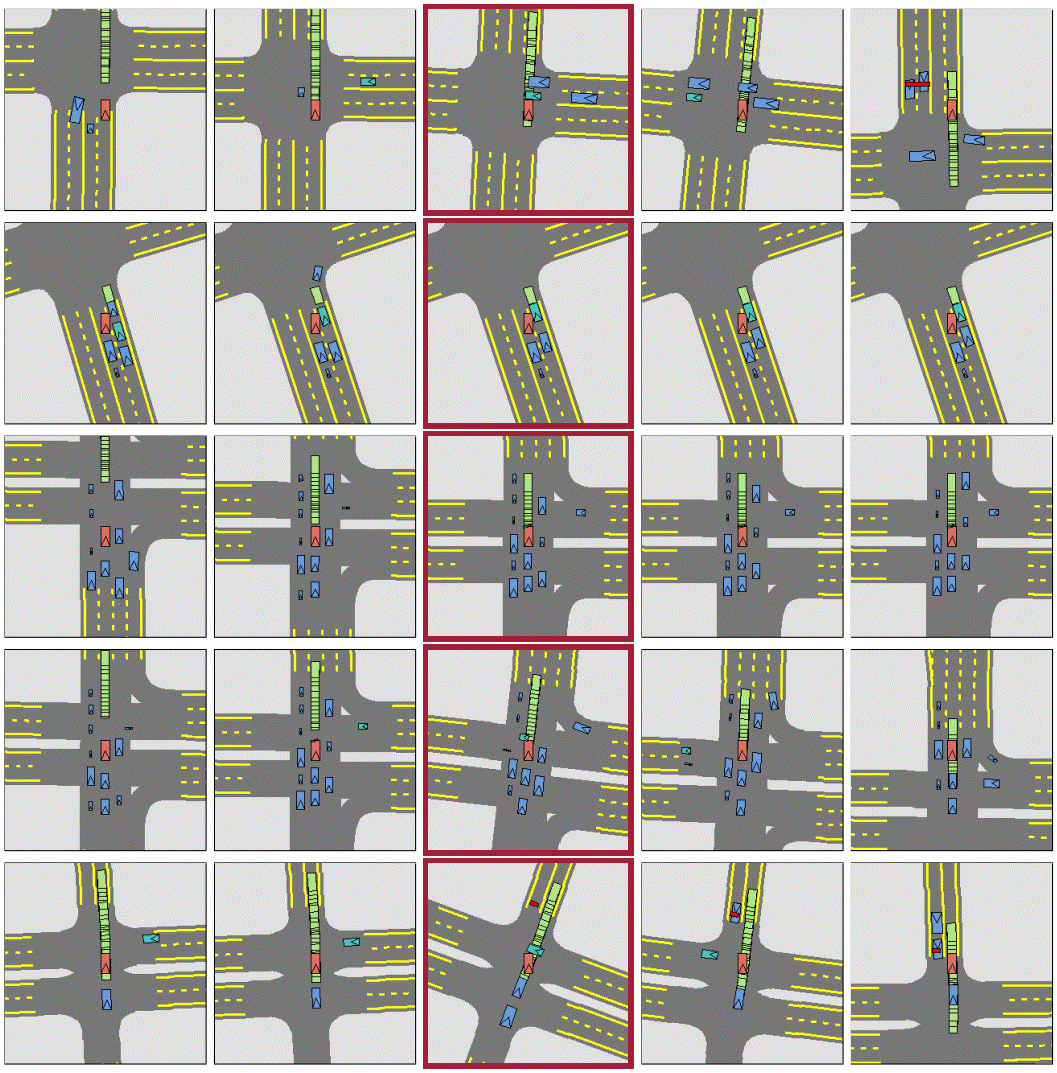


Figure 3.11: Cluster 2 of the entangled_mpc. The ego vehicle drives straight across an intersection and is hit from the right.

Figure 3.12 shows a cluster of 3 infractions on a highway. The scenes look very similar to the infractions we found for the *Conservative MPC* agent and the cause for the collision is again most likely the ending lane on the left which causes the neighbor vehicle to change to the lane on its right and merge into the ego vehicle.

Now let's have a look at clusters that were found for the *Autopilot* agent. Figure 3.13 shows a cluster containing 4 infractions that all happened at an intersection. In all cases the agent drives straight across the intersection and is hit by another vehicle from the right. The failure case that we found here is quite similar to the intersection scenes we investigated for the other two agents. It is quite interesting that the *Autopilot* agent also is involved in such a collision since it uses privileged input data.

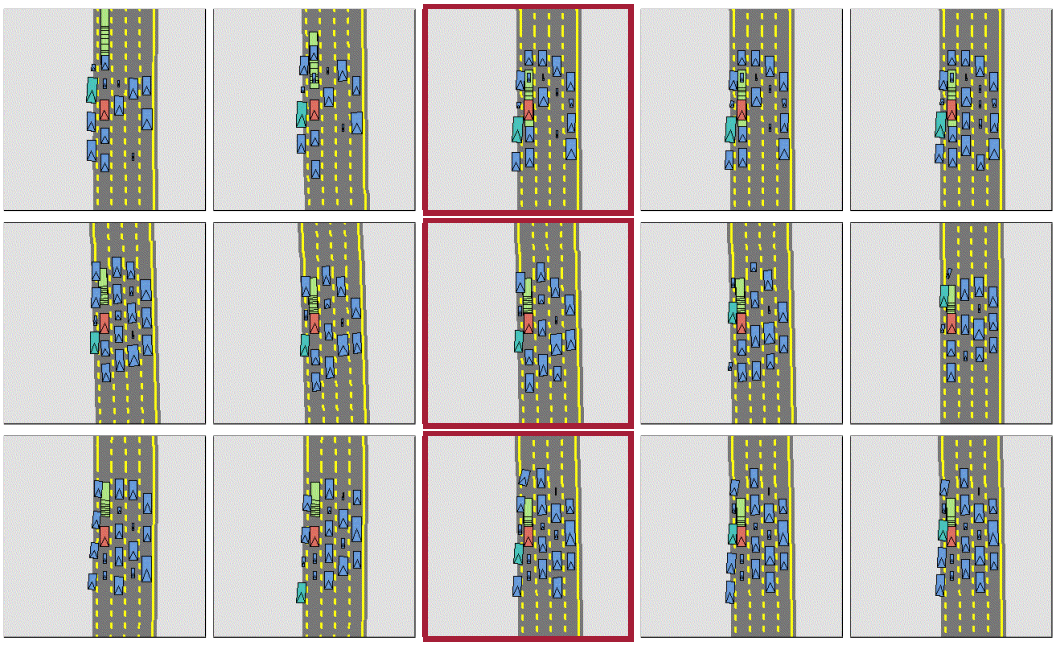


Figure 3.12: Cluster 3 of the entangled_mpc. Also for this agent we found the highway scenario where a car merges into the ego vehicle from the left because of the ending left lane.

Since the traffic at those intersections seem to be controlled by traffic lights this kind of collision could be caused because two lanes have a green light at the same time.

Figure 3.14 shows one of the larger clusters we found in this experiment containing 9 infractions (we didn't display all of them). All of them showing the exact same scenario that we are already familiar with. Just like with the other two agents, the ego vehicle is hit by another car from the left while it tries to switch the lane that is ending a few meters ahead. It is quite reasonable that the *Autopilot* is also involved in this type of infractions since it seemed to be completely caused by the other vehicle and even with privileged input data, the agent had no chance to avoid the collision in any way.

Now that we have looked at some examples of clusters we found for all agents we can discuss the results of this final experiment. First of all we could see that the clustering pipeline (executed with DBSCAN and the parameter and feature set we found in the previous experiments) was able to find groups of similar infractions that are likely to be cause by the same failure case. Of course some of the clusters we investigated contain some outliers or can further be split into 2 separate clusters but in the vast majority the infractions were grouped together correctly. We still need further investigation to really know what causes the agent (or the neighboring car) to fail in certain situations. However the proposed clusters are already a good hint for where to start this investigation and give a first impression of possible failure

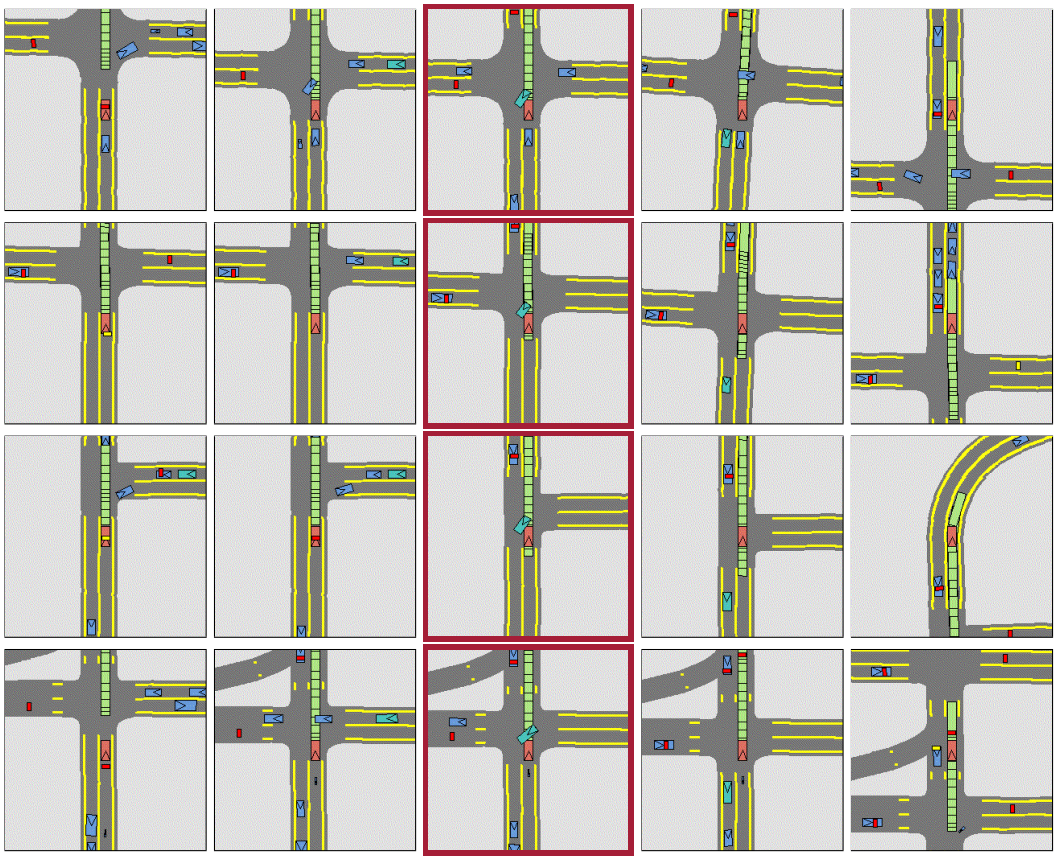


Figure 3.13: Cluster 1 of the autopilot_longest6. Another example of the case where the agent drives straight across the intersection and is hit by a vehicle coming from the right.

cases that we have to address.

Especially when we compare the infractions clusters between several different agents we can find very interesting insights. We already found two failure cases that happened in the evaluation of all three models which indicates that those cases are either caused by the CARLA environment itself or that we found general problems that are shared between architectures. For example we saw that all 3 agents were involved in collisions at intersections where the ego vehicle tries to drive straight and is hit by another vehicle from the right. As we said, we can't be sure what exactly causes those collisions without further investigation of the cases. There might be a problem with the traffic lights at some intersections allowing multiple lines to drive at the same time. Or all of the agents have problems to pay attention to the traffic from the right for some reason. A second example of a scenario that we found in the results of all agents was a highway scene where the very left lane ends which causes the vehicles to leave this lane. We saw many cases where this leads to a merge from

the left into the ego vehicle. In this case it is quite likely that the collision is not the fault of the agent. We couldn't find any obviously wrong behavior of the ego vehicle in any of the cases and it also had no chance to prevent the infraction.

All in all the results of this experiment show that by using the result parser in combination with the clustering pipeline we are able to extract meaningful features and find groups of infractions that help to find and investigate certain failure cases and to compare the performance of different agents in similar situations.

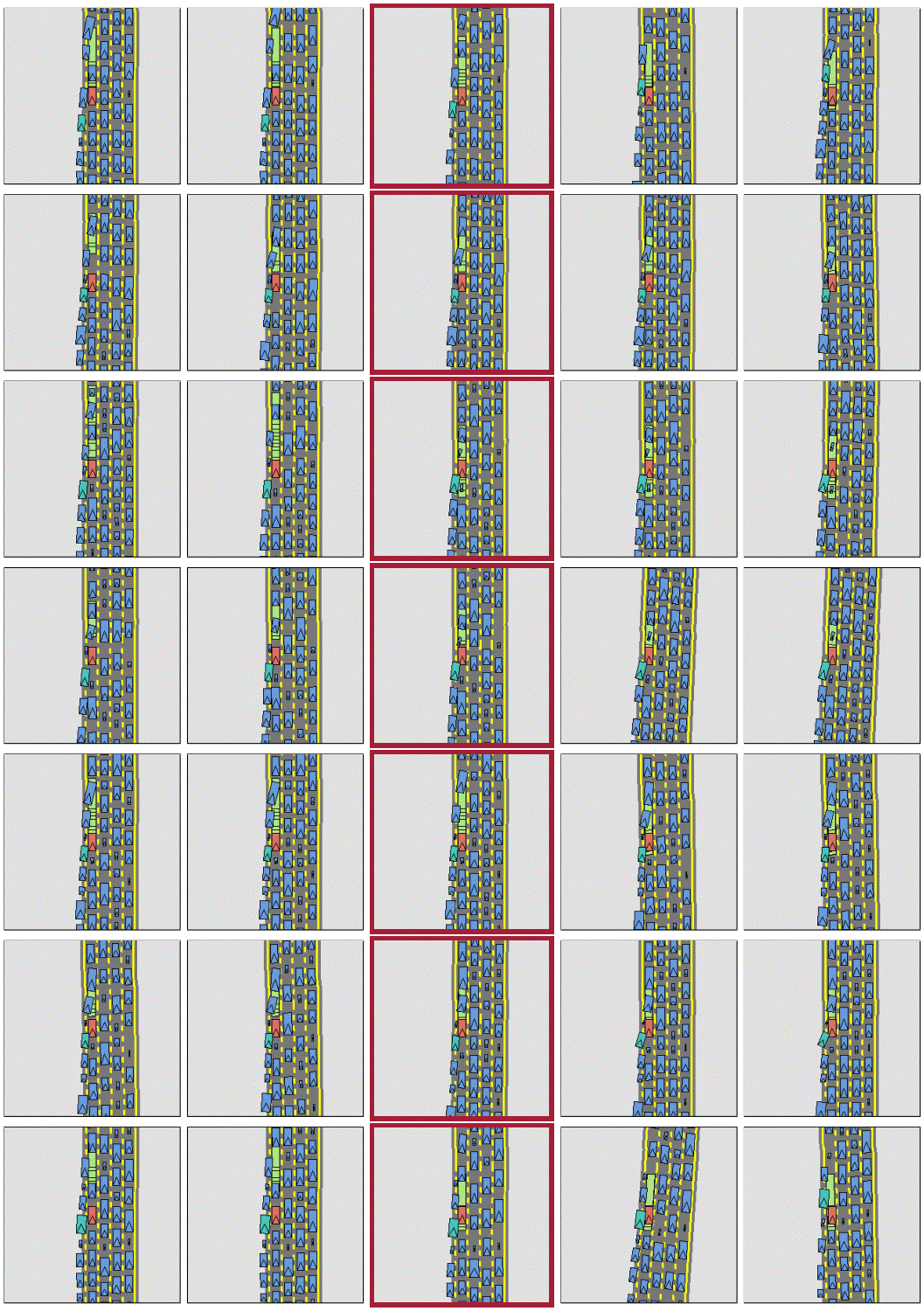


Figure 3.14: Cluster 2 of the autopilot_longest6. Again we found the collision caused by the ending left lane on the highway.

4 Discussion

We conclude this report with a short summary of our project and the results we got in our experiments, discussing advantages and flaws of the scripts and pipeline we implemented and a brief outlook on possible improvements and extension that could be part of subsequent projects.

4.1 Summary

In the beginning of our project, we have learned that evaluating a self-driving agent is a time-consuming work and finding problems and failure cases is not straight forward. Even when using an automated and simulated evaluation as it is possible in the CARLA simulator, going through all the entire recordings of such a evaluation run manually is a very inefficient and tedious process. Therefore as a main goal of this project we wanted to reduce those manual steps as much as possible and build a highly automated pipeline to enhance and simplify the whole process of evaluating and comparing different driving agents. The result parser script that we implemented and built together by using existing scripts and parts of code that was used in other projects can be executed directly after the evaluation in the CARLA environment is completed. Without further manual steps the different files that are created by the script can be used to get a first impression of the driving performance of our agent. The *results.csv* file contains important metrics like the overall driving score that is also used as the main criterion in the CARLA leaderboard, as well as a first overview of the number of infractions that the agent was involved in. By looking at the town maps that are rendered as images for every town the agent was driving through, we can get an even more detailed overview about where the infractions happen and which infraction type they belong to. One of the main parts of our contribution was the rendering of infraction clips. Instead of having to watch the entire recording of the evaluation run to find cases in which the agent was not able to handle a driving situation properly, those failure cases are now automatically extracted and rendered as short video clips of a few seconds length. By rendering the infractions in birds eye view and highlighting the ego vehicle, traffic lights and in case of collisions also the neighbor vehicle that was involved, it is now easier to quickly understand the scenario and investigate which problems have led to the infraction.

To further improve the evaluation process, as the second main part of our project we

implemented an automatic clustering pipeline that already outputs possible groups of similar infractions providing proposals for possible failure cases that happened several times. Again, no manual step is needed and the clustering pipeline can be automatically executed as soon as the result parser is finished. One important aspect that we wanted to consider during the entire implementation was to ensure that the pipeline can work with arbitrary clustering algorithms and parameters sets and it should very simple to change between them, add new algorithms or search for good parameters. We tried to cover this property by using abstract base classes that can be extended when implementing a specific algorithm. Now it is possible to use a whole set of different clustering algorithms and easily add or remove them. Since different clustering methods all have their specific advantages and disadvantages or are suited for datasets with different properties, using multiple algorithms and play around with their parameters result in different clusters that could reveal different failure cases or properties of the diving behavior of our agent.

4.2 Challenges

During the project we were confronted with several challenges that we had to tackle. In the following chapter we want briefly talk about the most interesting problems and how we tried to solve them.

4.2.1 Finding the Infraction Frame

The record files that are created during the evaluation and are used as an input for the result parser contain very detailed information about the entire state of the simulator in a close range to our ego vehicle. This state is saved in certain time intervals which leads to many timesteps for which we have the information that we need to render a BEV video clip, lets call those timesteps *frames* (as each timestep is rendered into one frame in the created video clip). In order to create a video clip that contains an infraction, we first have to find the frame in the recordings that corresponds to the timestep at which the infraction happened. However from the result file (the file that contains an overview over all infractions) we only know the exact position at which the infraction happened, not the correct frame. Therefore one big challenge during the implementation was to find this infraction frame and then rendering the video clip ranging from a few frames before the infraction frame and few frames after it.

The first naive approach was to just loop through the recordings and search for the frame in which the ego vehicle is located exactly at the position where the infraction happens. But since the those frame are only recorded in certain time intervals we can't check for the exact location since this is very likely not included in the frames we have but instead we had to check if the distance between the ego vehicle and the infraction position is smaller than a certain threshold. This threshold however leads

to a trade-off. If we set a value that is to small, it could happen that we overstep the correct frame since the threshold was never reached and we completely skip the infraction frame. If we choose the distance too high, we start the recording to early and the video clips will contain unimportant parts since the threshold will be reached at frame where the ego vehicle is still far away from the exact infraction position. Furthermore it was quite challenging to find a good value for this threshold since the tendency to overstep the infraction frame also depends on the speed of the ego vehicle. If the agent drives fast the difference between its position from one frame to the other is much higher as if the agent would drive slowly or even stand still. We tried to overcome this ambiguity by adding more conditions that has to be fulfilled in order to extract the current frame. For infraction types that involves other cars or town objects as for example *collision_vehicle* or *red_light* we additionally measure the distance to those objects and again used a threshold to check if we are close to the position where the infraction happened and also close to the object that is involved in this infraction. Although this improved the infraction frame extraction a lot there are still some scenarios where created video clips are not ideal or infractions are skipped. Therefore we also added a summary that is shown when the result parser is finished, containing a list of all infractions that were skipped because the corresponding frame could not be found. Depending on the current use case we can now increase the threshold if there are too many skipped infractions.

4.2.2 Dependence on the CARLA Server

The code part that is responsible for rendering the BEV videos was imported from another project. One problem with the existing implementation was that a permanent connection to a running instance of the CARLA server was needed in order to query the necessary information to render the street outline and other parts of the town maps. This has had consequence that the result parser could not be executed if the CARLA server was not installed and querying the server slowed down the script drastically. When observing this issue we found out that the queried information never changed and just depends on the town in which the agent was currently driving. To remove the permanent dependency on the CARLA server we implemented a method that is only executed if the script was executed for the first time. It tells the user to open an instance of the CARLA server, query all necessary information for all town maps and save it into files. For all further uses the script only has to read those files instead of querying the server. The files can also be shared to another computer which allows to run the script on systems where the CARLA server is not installed. The greatest advantage however was the huge run-time improvement (more than 20 times faster) as the parser does not have to query the server for every infraction anymore.

4.2.3 Measuring the Performance of Cluster Results

While implementing the clustering pipeline we encountered the problem that it was quite challenging to evaluate whether certain changes in code or features really improves the results. Just visualising the clusters and investigating the clusters by hand was too tedious. Especially when comparing different clustering algorithms or searching for parameters it was not tractable to evaluate the clustering manually. Therefore we first created kind of a ground truth for our dataset by manually dividing all infractions into clusters. Then we used the Jaccard Index as described in Chapter 3.2 to calculate how much a resulting set of clusters overlapped with our hand-clustered solution. This improved the implementation process a lot because after each change we could calculate a single metric that tells us if the changes improved the quality of the predicted clusters.

4.3 Experimental Results

In our experiments we compared different clustering algorithms, parameters and feature sets and selected the setup that showed the best results in terms of Jaccard index. We want to emphasize again that we did this to get a rough heuristic to choose one algorithm and parameters values to use in the final experiment. We highly recommend to try our different algorithms and feature sets as they may work much better for different scenarios. In Chapter 3.5.4 we visualized clusters that we found during our experiments. We demonstrated that our clustering pipeline was able to build reasonable and expressive groups of infractions that reveal general failure cases of a driving-agent. We also were able to find failure cases that happened independently of the used model. We found out that some infractions, for example the highway scenario, are hard or even impossible to avoid even when having access to privileged knowledge of the surrounding instead of sensor data. Other failure cases that did not happen to the autopilot for example collisions that happen during lane changes or on left turns as it was the case when evaluating the entangled_mpc model visualized certain problems of this model that can be investigated and addressed in further work. The qualitative results we visualized showed that the result parser and clustering pipeline work quite well and produce meaning full results that help to get a feeling of which driving situations are challenging for a trained agent. We also showed that the clustering results are of course not perfect and that there are some examples of infractions that were assigned to the wrong cluster or accidentally were determined as noise. Therefore, the clustered results should always be treated with a grain of salt and are just meant to give an overview and a first impression of possible scenarios and failure cases that we should investigate further.

As a conclusion we can say that the experimental results demonstrated that the implemented result parser and clustering pipeline work well and are useful tools that can support the evaluation process in the future. The experiments also revealed

certain insights that already could be used to avoid certain infractions and improve the overall performance of the investigated models.

4.4 Outlook and possible improvements

In the last chapter of this report we want to list a few ideas for future work and possibilities for further development of the evaluation tools.

- **Adding more features.** The result of clustering algorithms highly depend on the set of features we use to describe the infractions. The list of features that we used for our experiments is just one possible choice and it could be extended by further features. As we showed in Chapter 3.5.3 the orientation and relative position of the two cars that are involved in the collision were quite important for the predicted clusters and the Jaccard Index. This makes sense since the cluster we found could be expressed quite well by only looking at the current movement of the ego vehicle (for example in case of straight driving, left turns, right turns etc.). By adding additional features, we could also find clusters that depend on other properties of the collision. For example, we could calculate where the bounding boxes of the two vehicles intersected to divide the intersections not by the movement of the vehicles but by the way they collided. This type of clustering was also done by Rempe et al. [RPG+21]. Figure 4.1 shows 6 different collision types that were found in this paper by clustering collisions in a virtual driving environment by the way the two vehicles collided.

Figure 4.1: Different collision types found by [RPG+21]. The red arrows indicates the direction from which the other car crashed into the ego vehicle.

  Until now we only considered features from the ego vehicle and the neighboring vehicle that collided with it. By adding information from all cars in the current surrounding we could also cluster certain scenarios like traffic jams.

- **Generate video of infractions directly in CARLA.** Bird's eye view videos are great to get an overview of a driving scenario. However this representation is also quite minimalistic and we might loose some information that would be important to understand why our agent failed. Maybe the agent ignored a stop sign because of the heavy rainfall or because the sun hit the camera in an unfavorable angle and influenced the object detection. Factors like weather, lighting conditions or obstacles like buildings are not rendered in the BEV

videos but could be important for investigating failure cases. Therefore, it could be helpful extension to render the infractions not only in BEV but also directly in the CARLA environment to exactly replay the scenario as it was captured by the cameras and sensors of the ego vehicle. Since the record files contains the all necessary information to rebuild any state of during the evaluation it would also be possible to recreate and replay them in the CARLA simulator.

- **Use the failure cases as unit tests for new agents.** The failure cases that we found are not only useful for retraining or improving the current agent but can also be used to test new agents. If we found a scenario which was challenging for an agent it is quite likely that also other trained agents in the future may have problems with those situations. To reduce the evaluation time we could evaluate a new agent directly on those scenarios to see if it can handle them. By doing this we get challenging test scenarios without having to do an entire evaluation run that may take several hours. Further we can directly compare how several agents that have different model architectures behave in the same scenario which can also lead to interesting and important insights.
- **Query CARLA autopilot to create additional highly relevant training data.** Finding certain failure cases is quite useful but we also want to find ways how to overcome those problems we discovered. As many state-of-the-art agents for self-driving were trained using imitation learning, this means that the agent learns by trying to imitate the behavior of an expert driver in all possible scenarios. In our case the autopilot is used to record this expert data. One problem of this approach is that challenging or dangerous situations are very rare and therefore it is hard to get enough training data in such situations in order to teach the agent how to behave in those scenarios. Now that we found many challenging situations (all the infractions that occurred during the evaluations) we can use them to create additional training data for critical scenarios. We simply have to let the autopilot drive through the scenarios that lead to infractions and record its driving behavior. Next, we can retrain our models on this data to train them to handle those situations in the future.

Bibliography

[ABKS99] Mihael Ankerst, Markus Breunig, Hans-Peter Kriegel, and Joerg Sander. Optics: Ordering points to identify the clustering structure. *Sigmod Record*, 28:49–60, 06 1999.

[AL21] Hesham Alghodhaifi and Sridhar Lakshmanan. Autonomous vehicle evaluation: A comprehensive survey on modeling and simulation approaches. *IEEE Access*, PP:1–1, 11 2021.

[BP95] J.C. Bezdek and N.R. Pal. Cluster validation with generalized dunn's indices. In *Proceedings 1995 Second New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems*, pages 190–193, 1995.

[CB04] Eduardo Camacho and Carlos Bordons. *Model Predictive Control*, volume 13. 01 2004.

[Che95] Yizong Cheng. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8):790–799, 1995.

[CK22] Dian Chen and Philipp Krähenbühl. Learning from all vehicles, 2022.

[CPJ+22] Kashyap Chitta, Aditya Prakash, Bernhard Jaeger, Zehao Yu, Katrin Renz, and Andreas Geiger. Transfuser: Imitation with transformer-based sensor fusion for autonomous driving, 2022.

[CTHM21] Raphael Chekroun, Marin Toromanoff, Sascha Hornauer, and Fabien Moutarde. Gri: General reinforced imitation and its application to vision-based autonomous driving, 2021.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[HRC+22] Niklas Hanselmann, Katrin Renz, Kashyap Chitta, Apratim Bhattacharyya, and Andreas Geiger. King: Generating safety-critical driving scenarios for robust imitation via kinematics gradients, 2022.

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. *The New Phytologist*, 11(2):37–50, 1912.

[Mac67] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[MSB+17] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[Mur83] F. Murtagh. A Survey of Recent Advances in Hierarchical Clustering Algorithms. *The Computer Journal*, 26(4):354–359, 11 1983.

[Rou87] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[RPG+21] Davis Rempe, Jonah Philion, Leonidas J. Guibas, Sanja Fidler, and Or Litany. Generating useful accident-prone driving scenarios via a learned traffic prior, 2021.

[SAIR11] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, 2011.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[SSE+17] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: Why and how you should (still) use dbscan. *ACM Trans. Database Syst.*, 42(3), jul 2017.

[SWC+22] Hao Shao, Letian Wang, RuoBing Chen, Hongsheng Li, and Yu Liu. Safety-enhanced autonomous driving using interpretable sensor fusion transformer, 2022.

[WJC+22] Penghao Wu, Xiaosong Jia, Li Chen, Junchi Yan, Hongyang Li, and Yu Qiao. Trajectory-guided control prediction for end-to-end autonomous driving: A simple yet strong baseline, 2022.