Modelling of complex traffic scenarios around Autonomous Vehicles through states and state transitions of traffic objects

Henrik Håkansson, Katarina Tran, Devosmita Chatterjee

January 17, 2020

Abstract

The paper aims to find a way to identify all possible kinds of complex traffic scenarios around vehicles. We propose a states-and-transitions-model for describing scenarios. In the model surrounding traffic objects are described by occupation grids of size 3x3 cells, one grid for each type of traffic object. Grids are translated into binary matrices where 1 denotes occupation of nearby traffic object. Also attributes such as the vehicle's speed and angular movement are used. The model is evaluated by analysing sequences of states computed from real traffic data recorded in an intersection. We conclude that using clustering we are able to distinguish different scenarios represented by the model.

Contents

1 Introduction

Ensuring safety of operation is one of biggest challenges for market introduction of autonomous driving (AD) vehicles. AD vehicle must manage with any traffic scenario that can possibly happen. Therefore, to verify AD, the comprehensive set of traffic scenarios needs to be identified first. In order to determine between what should be interpreted as similar scenarios, and what should not, there is need for a mathematical model. In this project we propose a states-and-transition-model and investigate its capability of distinguish different scenarios by applying the model on field data from a stationary surveillance sensor.

The model we propose focus on the scenario for one specific vehicle, which we believe might consist of two main parts: attributes for the movement of the vehicle and attributes describing of other nearby traffic objects. The first mentioned category of attributes are easy to find in the data consisting of trajectories of traffic objects. The second category may leave room for creativity and a wide range of different approaches. We try to simplify the neighbourhood by using a grid of 3x3 cells surrounding the vehicle, and identify relevant objects as the ones occupying some cell of the grid. We will investigate whether this representation is useful for extracting a sensible set of scenarios and how the size of such grids will affect those extracted scenarios.

The paper is organized as follows. The data set that has been used for evaluating the model is presented in Section 2. Section 3 consists of the definition and formulation of the model. In section 5 clustering techniques used for analysing sequences of states are presented. Section 5.3 describes implementation of computation of field data into states represented by the model and how analysis on these have been carried out. The results after implementing clustering algorithms are analyzed in section 6 Finally, we discuss the outcomes and give some conclusions in section 7 and 8.

The methodology used in the project is briefly described in Figure 1.

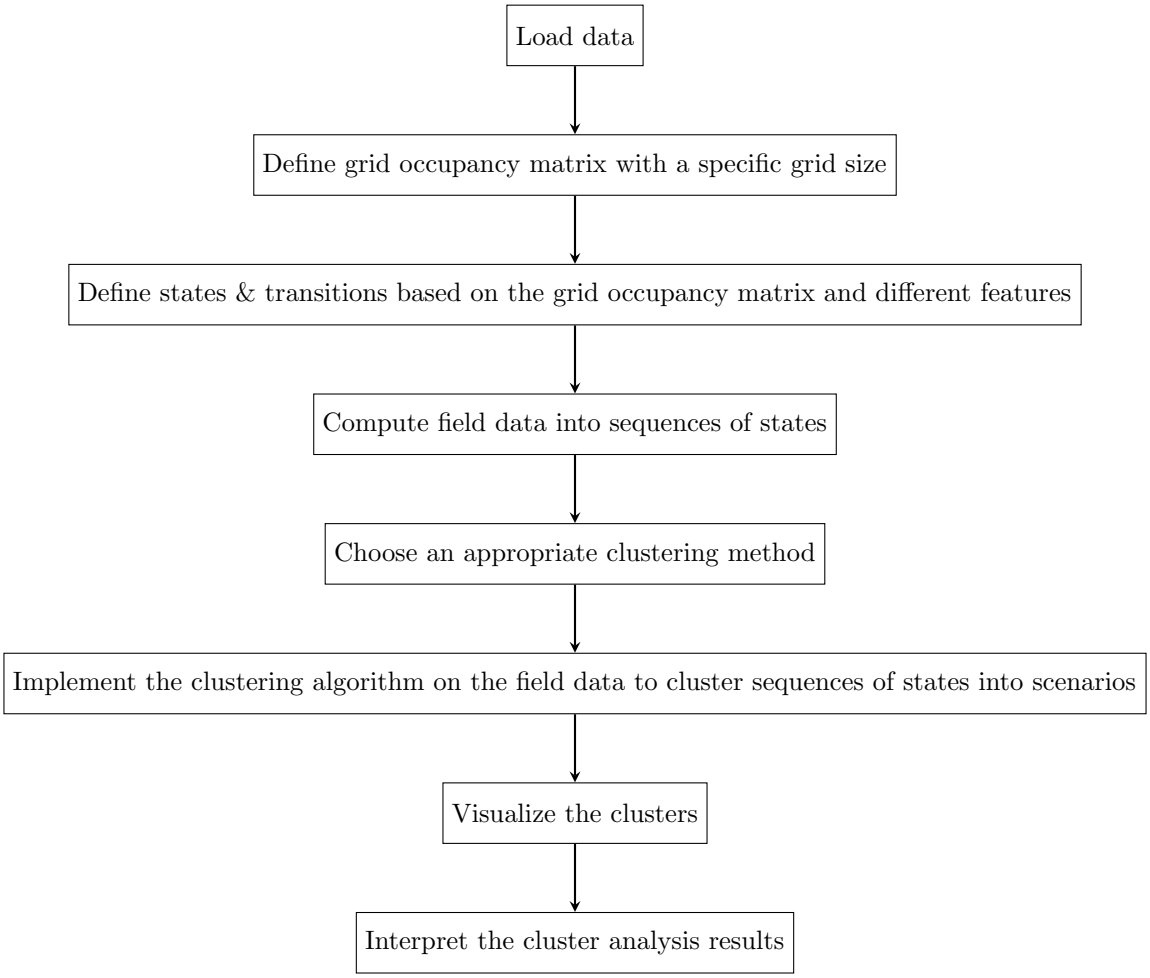

Figure 1: Schematic overview of the method

2 Data set

The data set, provided by Viscando Traffic Systems AB, contains trajectories from traffic objects in the intersection between the streets Frederiks gate and Kristian IVs gate in Oslo, recorded in April and May 2017. Each traffic object is classified to one of the types pedestrian, bicyclist, vehicle or heavy vehicle. It includes information like object id (ID), time (Time), x, y, Speed, Type, Estimated. Each traffic object is denoted by a particular ID. Time represents the particular time at which a specific object is tracked by the camera. x, y denotes the deviation from the reference point on the map provided by Viscando. Latitudinal and longitudinal coordinates were calculated from pairs of x-y-data for each traffic object with respect at different times. The x-y-data were first rotated by an angle of $\frac{3\pi}{2}$ by using the rotation matrix. Latitudinal and longitudinal coordinates are given by

$$[lat, long] = (c_y \cdot y + lat_{ref}, c_x \cdot x + long_{ref}).$$

where c_x and c_y are constants and [lat_{ref}, $long_{ref}$] is the approximated lat-long coordinates at the reference point. The lat-long coordinates were used to plot trajectories in a final map through Mapbox. Speed represents the speed of the specific traffic object at a particular position and time. Type denotes the category of the specific traffic object. Estimated is a binary column where 1 implies that the the object is not tracked by the camera and thus it is estimated while 0 implies that the object is tracked by the camera and hence it is not estimated. Angle is computed by the formula $\theta = tan^{-1}\big(\frac{long}{lat}\big)$.

3 States and transitions model

The core of our way to attack the problem is to use a limited number of features to describe the state of a traffic object in a certain time. Features that are included in our model are relative positions of nearby traffic objects, speed and acceleration of the traffic object, turning angle of movement of the traffic object and state of lane.
A state is defined as the surrounding situation of each traffic object at a specific time. A scenario is described by a sequence of such states.

3.1 Grid and occupation matrices

At each time for each traffic object occupation matrices are computed to describe its interaction with other traffic objects. For each type of traffic object (pedestrian, bicyclist, vehicle, heavy vehicle) there is such occupation matrix computed when determining the state. In this section the traffic object which we aim to determine the state of is called the host object, while other traffic objects that are taken into account when computing occupation matrices are called attending objects.

3.1.1 Occupation matrices

The 4 different occupation matrices are denoted: P (pedestrian), B (bicyclist), V (vehicles) and H (heavy vehicles). The occupation matrices for different host object and/or at different times may differ from each other.

An occupation matrix for one of the 4 traffic object types is a 3x3 matrix whose entries corresponds the positions of attending objects of the given relative to the movement of the host object. Entries in this matrix denote the presence of attending objects of the given type close to the host object. For the host object at the particular time a 3x3 grid is aligned around the host object (further explained in section 3.1.2) The entries of an occupation matrix for a traffic object type is 1 for an entry at row i and column j if there is an attending object (or the host object itself) of the given type present in the corresponding cell at row i and column j of the aligned grid. If there are no objects of the given type present in a cell, the value of the entry is set to 0.

Example 1: a bicyclist present at cell $(1, 1)$ gives 1 for the entry $(1, 1)$ in the bicyclist occupation matrix.

Example 2: no heavy vehicles in the cell $(1, 2)$ gives 0 for the entry $(1, 2)$ in the heavy vehicle occupation matrix.

3.1.2 Grid alignment

At each time for each traffic object a grid with 3 rows and 3 columns is aligned according to the movement of the traffic object. This direction is given by the vector between two recorded positions of the host object. When computing the state for the host object at some time we use the two latest recordings for computing the movement direction. The columns of the grid are aligned along this direction and the rows orthogonal to this direction. The center of the grid is located at the coordinates of the last recording of the host object.

3.1.3 Compute occupation matrix by using the grid

All traffic objects that are inside the grid are used when computing the occupation matrix. The grid is indexed by row and column ascending in the opposite of the movement direction for the rows and left to right of the movement direction for the

columns. For example, row 1 corresponds to cells in front of the host object and column 2 corresponds to the cells to the right of the host object.

Each cell in the grid corresponds to the entry of the matrix with the same index, meaning that if a traffic object of some type is within a cell of the grid the symbol explained in section 3.1.1 of its type will be appended to the corresponding entry in the occupation matrix.

3.1.4 Examples of computation of occupation matrix

In these 3 examples there are 4 traffic objects: the host object is a car (vehicle), and the attending objects are: one pedestrian, one bicyclist and another car. With velocities according to those in Figure 2 the following occupation matrices are retrieved:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, V = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, H = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

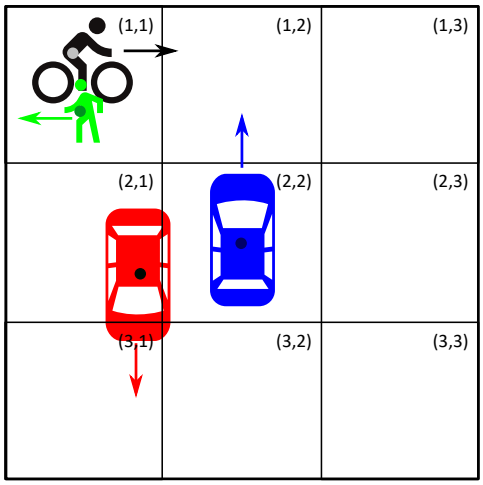

Figure 2: *Example 1 of grid for with the 4 traffic objects. The blue car is the host vehicle, other objects are attending objects. Dots symbolise the point of center of mass of the traffic objects, which is where the coordinates in our data origins from. Arrows denote the movement direction of each traffic object. The cell indices are written in the upper right corner of each cell.*

If all the movement directions of the attending objects are the opposite directions compared to those in Figure 2, which is visualised in Figure 3, we get the same occupation matrices because movement directions are ignored when computing the occupation matrix.

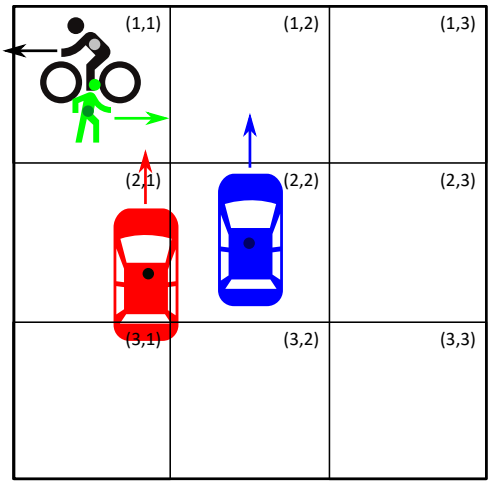

Figure 3: *Example 2 of grid for with the 4 traffic objects. Meaning of symbols same as in Figure 2.*

If we instead set the direction of the host object to the opposite direction compared to 2 we end up in matrices mirrored by the diagonal. This example is visualised in 4 and the occupation matrices becomes:

$$P = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, V = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, H = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

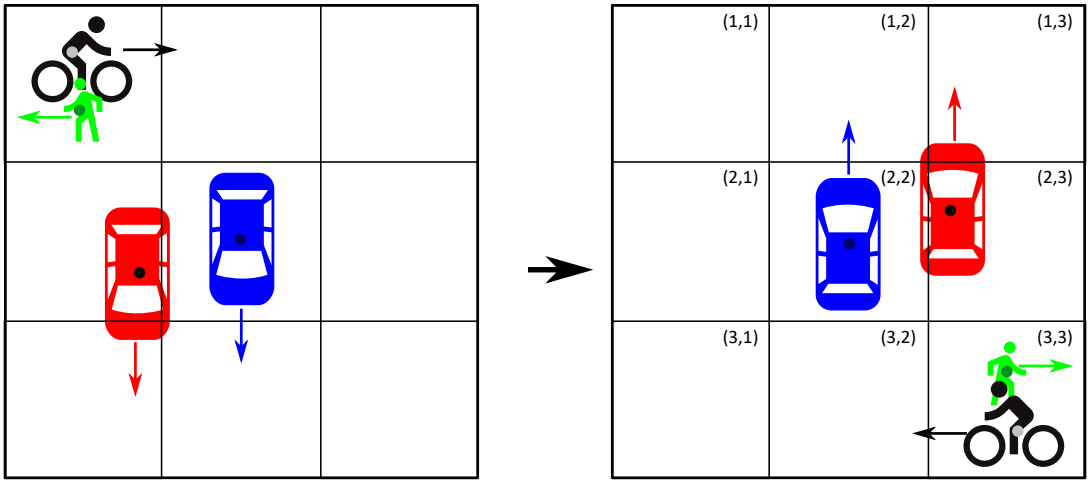

Figure 4: *Example 1 of grid for with the 4 traffic objects. Meaning of symbols same as in Figure 2. Left grid is not rotated correctly in order to compute occupation matrix, while right grid is the same set up of traffic objects but grid is rotated correctly.*

3.2 State vector

The state for a particular traffic object at a particular time can be represented by a vector that utilizes the idea presented earlier in this section. This state vector S_{TO} for a specific traffic object (TO) is defined by

$$S_{TO} = [P_{TO}, B_{TO}, V_{TO}, H_{TO}, s_{TO}, \Delta\theta_{TO}].$$

P_{TO}, B_{TO}, V_{TO} & H_{TO} represent the occupation matrices for different categories of traffic objects - pedestrians, bicyclists, vehicles and heavy vehicles, respectively. s_{TO} denotes the speed of the specific traffic object provided in the data set. $\Delta\theta_{TO}$ represents the change of angle for a specific traffic object. $\Delta\theta_{TO}$ is given by the formula

$$\Delta\theta_{TO} = \frac{\Delta\theta}{\Delta t} = \frac{\theta_2 - \theta_1}{t_2 - t_0}$$

where θ_2 & θ_1 denote the respective angular movements per second of the latest recordings at timesteps t_2 & t_1 and t_1 & t_0.

4 Data exploration and visualisation

There can be many types of scenarios in traffic, in this model a scenario is composed of sequences of state vectors. One possible way to investigate whether the model can be used to describe different traffic scenarios is to find distinct clusters when plotting sequences of parameters in state vector (calculated from data provided by Viscando) in a 3D plot. Each cluster would then represent a specific type of scenario. The focus in this section is not to find every single one of them but rather to focus on a few critical scenarios and analysing the changes of parameters for each type.

4.1 Data cleaning

Since the focus was on scenarios that were the most probable, odd trajectories that deviated very much from the rest or too short were considered as outliers and were removed (as good as possible). This makes it easier to distinguish between clusters in the 3D plot.

Outliers were filtered out by matrix rotation and then setting some criteria on the x-y-coordinates. Left figure (5) shows all data tracks for vehicles before data filtering. Right figure (5) shows the filtered out tracks, in this case a right turn track.

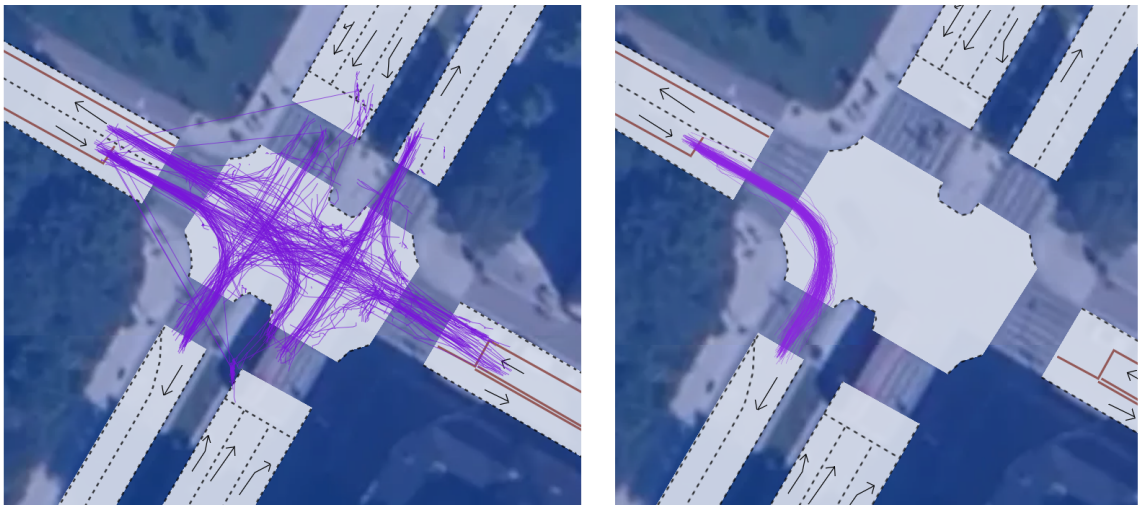

Figure 5: *left picture shows tracks for all data collected before filtering. Right picture shows tracks for right turn after filtering.*

4.2 Plotting of state vector parameters

4.2.1 Case: right, straight and left paths

Let's start with some simple scenarios for a vehicle: turning left, turning right and driving straight. scenarios. Figure (6) and (7) shows trajectories of left, right and straight paths that were used in the 3D plotting.

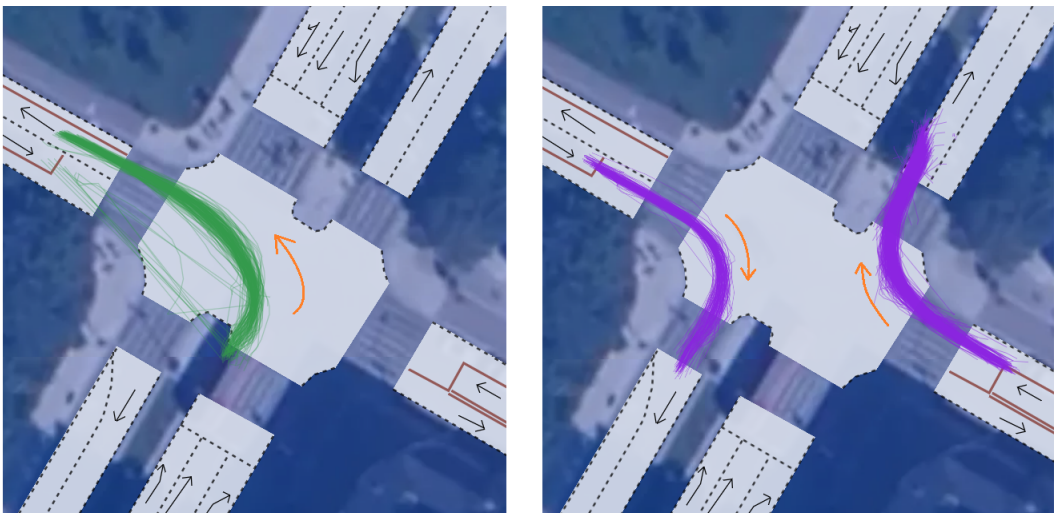

Figure 6: *pictures showing tracks for left and right turn.*

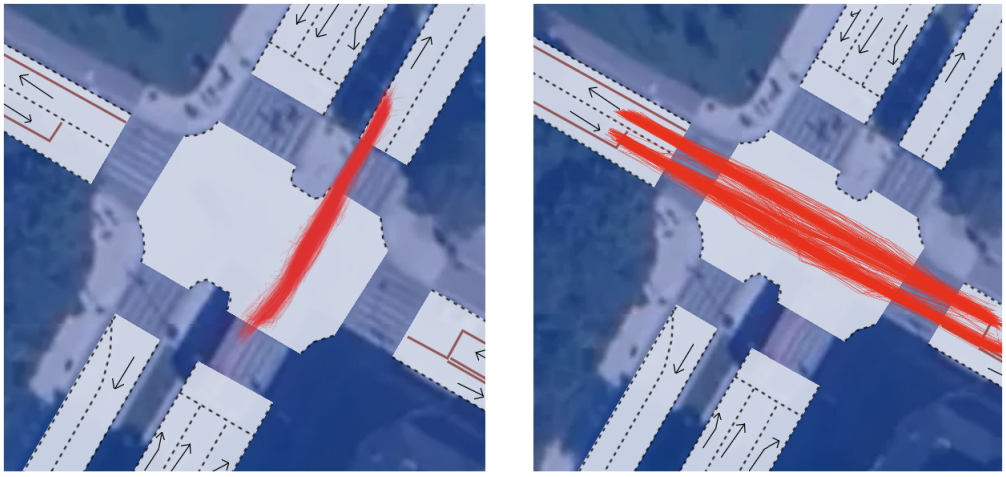

Figure 7: *Tracks for vehicles driving in a straight direction, used for clustering.*

The result from plotting sequences of state vectors with the parameters: cumulative theta, time and speed is shown in figure (8). The colors are there to indicates which type of scenario each plot line belongs to.

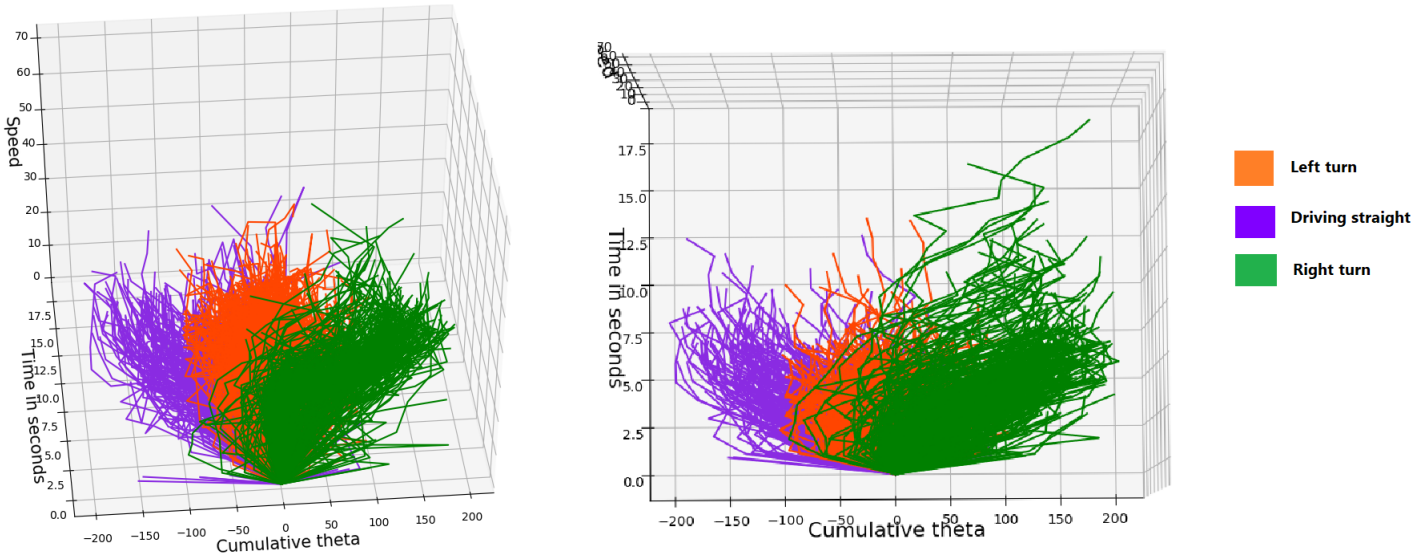

Figure 8: *clustering of left, straight and right trajectories.*

4.3 Plotting of scenarios involving occupation matrix

4.3.1 When only two traffic objects are involved

In case when only two traffic objects are involved in the scenario (host and another object) the dimension of state vector is reduced to four : cumulative theta, time, speed and the cell position at which the occupation matrix is occupied. If each cell in occupation matrix is given a specific color for distinction then all the parameters can be visualised in a 3D plot, as shown in figure(9). Here the scenario is: car driving to the right and stops for pedestrian. The positions of the scatter points indicate that a pedestrian has been detected in front of the vehicle in cell 'P12'. The green line plots shows the state sequences for each vehicle.

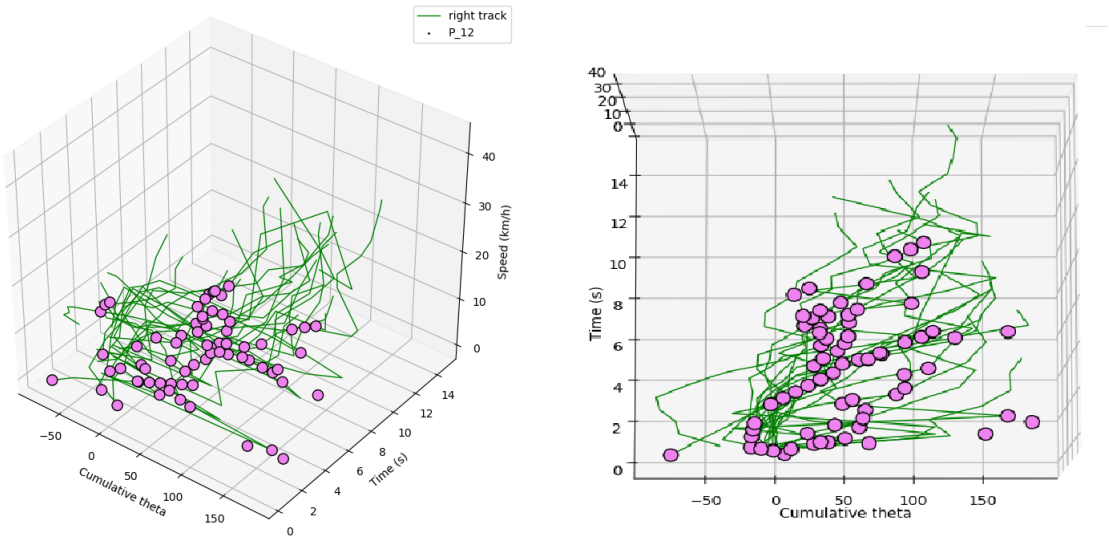

Figure 9: *Plot showing scenario: vehicle turning right and stops for pedestrian. The scatter points indicates that a pedestrian has been detected in front of the vehicle.*

5 Applied clustering techniques

Clustering is a common method for statistical data analysis, which is used in various fields such as machine learning, data mining, pattern recognition, image analysis and bioinformatics. Clustering is the process of organizing similar objects into different groups. Clustering is the most important unsupervised learning algorithm since it deals with unlabeled data.

5.1 K-means

The k-means algorithm is used to partition the set of points in the data into k clusters where each cluster has a point (not necessarily found in data) that is called its centroid. The objective of the algorithm is to minimise the summed distance from each point to the centroid of its cluster. To compute this an iteration of two steps is performed until convergence[1]:

1. Given a set of k centroids (starting with a random), all points are partitioned into the cluster whose centroid is closest to this point
2. When all points are assigned to some cluster, the centroids of each cluster are recomputed to be the mean point of all points in a cluster. The new centroids are then used once again in the first step.

This is an approximative compuational method and often the algorithm returns a local minima instead of the global. Another problem is that the original algorihtm might converge quite slow, but that can reduced by using the mini-batch approach [2]. In that variant of the algorithm a random sample of points are used in the first step of the iteration described above. For each iteration a new sample is taken, while computed centroids are saved for nextcoming iteration.

5.2 DBSCAN algorithm

One problem with k-means is that if little is known about the data before the analysis, it might be difficult to find a reasonable value for the number of clusters. Density-Based Spatial Clustering of Applications with Noise (DBSCAN)[3] is a clustering method that enables finding an unspecified number of clusters of arbritrary size. There are only two parameters required: the neighbourhood radius ϵ and the minimum number of points in a cluster $minPts$. For this work, the algorithm might be useful because little is known by the distribution, and it can be used for filter out noisy data that is too dissimilar from other data.

The point p_1 is direct density-reachable from the point p_2 if the distance between p_1 and p_2 is less or equal than ϵ and there exists at least $minPts$ other points within a radius of ϵ from p_2. A core point is a point that has at least $minPts$ other points within a radius of ϵ from it, e.g. if p_1 is direct density-reachable from p_2 we know that at least p_2 has to be a core point (though p_1 can also be a core point, but it does not have to). They are density-reachable if there is a chain of points where all neighbours in the chain are direct density-reachable and the ends of the chain are p_1 and p_2. Clusters are created by connecting core points with all its density-reachable points. Points that does not belong to any cluster are regarded as noise.

5.3 Method and implementation

Using the data from the intersection in Oslo, states have been computed for vehicles at discretized time steps. There have been a number of considerations on how to implement and analyse the results such as the time between state computations, handling of data of low quality and what size the grid should have.

5.4 Computation of field data into sequences of states

Using data of trajectories from pedestrians, bicyclists, vehicles and heavy vehicles in an intersection in Oslo sequences of states have been computed using a program written in Python. At specific times the state vector was computed for each traffic object present in the intersection at each time. To limit the scope of this project only states for vehicles were computed at these times, called state computation times.

An example of the computation of states is visualised in Figure 10 where state computations are performed periodically with an interval of d seconds. When computing the state for one of the vehicles, only the most recent recordings for each traffic object is used. When computing the occupation matrices only the most recent position for each traffic object are used. That means for each vehicle the recording closest to the left of a dashed vertical bar are the recording that is used when computing occupation matrices.

Because the state computations are performed recurrently with a constant interval, independently of the differences of times between recordings of individual vehicles, there might be more than one recordings for one traffic object between a pair of state computations. In the figure there is an example of this for the blue car in the middle between $t+d$ and $t+2d$ in Figure 10

There might also be differences in time between recordings of one vehicle that are longer than the interval between the state computations, as for the yellow car at state computation $t+3d$ and $t+4d$ in Figure 10. This results in uncertainty of the vehicle's behaviour in forthcoming state computations, why such cases are handled in a specific way described in section 5.4.2.

Traffic objects have been regarded to be present in the intersection if the first recording of the object occurred after the state computation time and if the last recording of the object were ahead of the state computation time or occurred within a defined time interval before the state computation time. For each of the present objects at the state computation time the

state vector is computed. In the example in Figure 10 the red vehicle is not present at the two first state computations at t and $t + d$, because its first recording occurs after these times. The blue vehicle is not present at the last state computation at time $t + 6d$, because its last recording happened more than d seconds before the state computation.

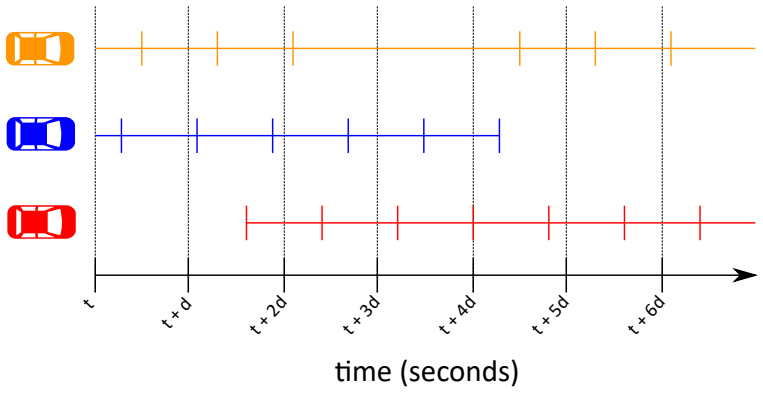

Figure 10: *Schematic example of state computations with a difference of d seconds between each state computation involving 3 vehicles: the red car at the bottom, the blue car in the middle and the orange car at the top. The colored lines indicates where there are recordings for each vehicle, corresponding to vehicle of same alignment and color. White gap of blue line to the right of the blue vehicle means there are no more recordings in future for this vehicle, similarly the white gap at left the left for the red vehicle means there are no more recordings for this car in the past. Time goes from t seconds to $t+6d$ seconds on the horizontal axis. The small vertical lines (colored in red, blue or orange) corresponds to times of recordings of the vehicle of the same color. The black vertical lines (from bottom to the top a $t, t + d, t + 2d \ldots$) denote the state computation times, e.g. when states are supposed to be computed for each vehicle using the most recent recordings.*

5.4.1 Length of time step

The length of the state computation period, d in Figure 10, should be a time interval long enough to make sure the vast majority of traffic objects are recorded between each pair of state computations. But the length of the period should also not be too long in order to still be a fair approximation of continuous time.

When plotting a histogram with a sample of time differences between recordings of the same traffic object, shown in Figure 11, we see that the difference is rarely larger than 0.25-0.3 seconds. For that reason, we have set the length of the state computation period d to 0.3 seconds.

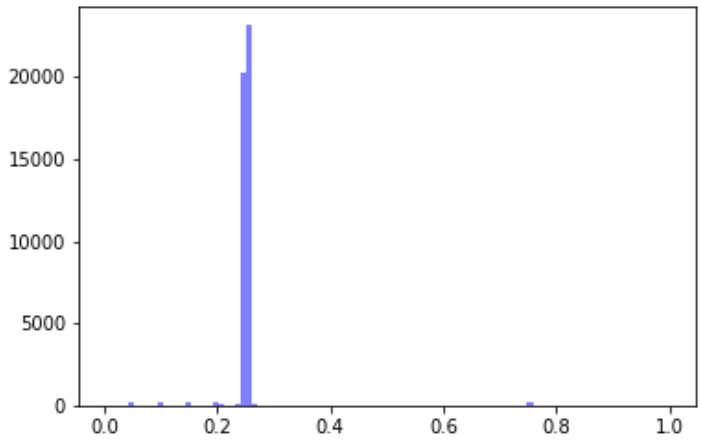

Figure 11: *Histogram of time differences between recordings of the same traffic objects. All differences are measured between recordings of the same traffic object, but there are many different traffic objects used. Outliers with a time difference greater than 1 are ignored in this plot, though such are too rare to be visible in this kind of plot.*

5.4.2 Filtering fragmentary trajectories

Even though the vast majority of the recordings for some object are updated within 0.3 seconds, there are also a lot of traffic object for which there suddenly appears a larger gaps between recordings. An example of such trajectory the yellow car in Figure 10 and a real example from the data is visualised in Figure 12 below.

When such gap appeared the object was regarded to be *invalid*, and by that not allowed to be used for state computations for state computation times within the interval with the lack of sufficiently frequent updates of recordings. This was implemented in the following way for each present traffic object and each state computation:

1. The difference between the time of the state computation and the recording of the traffic nearest in time before the state computation of the was checked. If this difference was greater than 0.3 seconds, the object was regarded as invalid.
2. If the object was invalid, it could not be used for computing the state.
3. If the object was not invalid, it could be used for computing the state. But if any of the attending objects within the grid were invalid, the computation was discontinued.

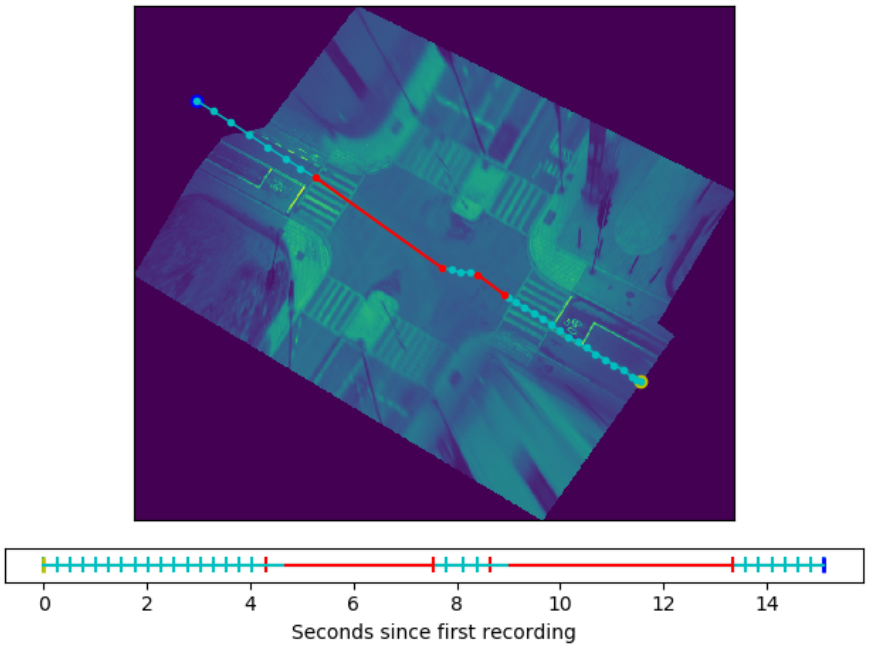

Figure 12: *Example of a trajectory with invalid fragments, using a maximally allowed time gap of 0.3 seconds. Upper panel shows positions from the traffic object. Yellow node indicates the first recording, blue node indicates the last recording of the object. Red sections indicates that the time between those recordings is larger than the maximally allowed time gap. The lower panel corresponds to same trajectory, but visualises the time gaps between the recordings. All times are relative to the first recording (time=0, at the yellow vertical bar to the left). The red vertical bars corresponds to the recordings between which sections are red-marked in the upper panel. The red horizontal line shows the time interval when this traffic object is regarded as invalid (starts the maximally allowed time gap, 0.3 seconds, after the last recording before the gap and ends at the next recording). Blue vertical bar is the last recording of the traffic object.*

5.5 Extract and analyse scenarios only using occupation matrices

In order to evaluate the capabilities of the occupation matrices, sequences of states have been clustered and analysed by only using the occupation matrices (ignoring speed and angular movement). Even though the data consists of trajectories from the whole intersection, these experiments have been limited to a smaller area shown in Figure 15. The reasons why this region is selected for this purpose are several:

- Analysis of the basic performance of the model will be simplified, because the number of possible scenarios are much fewer than if the whole intersection is taken into account.
- The area contains one single lane that goes straight in one direction, meaning at least angular movement is not needed to separate scenarios.
- Inside this area the expectation is that all 4 kinds of traffic objects types are represented. This can also be seen in the data and visualised in Figure 15.

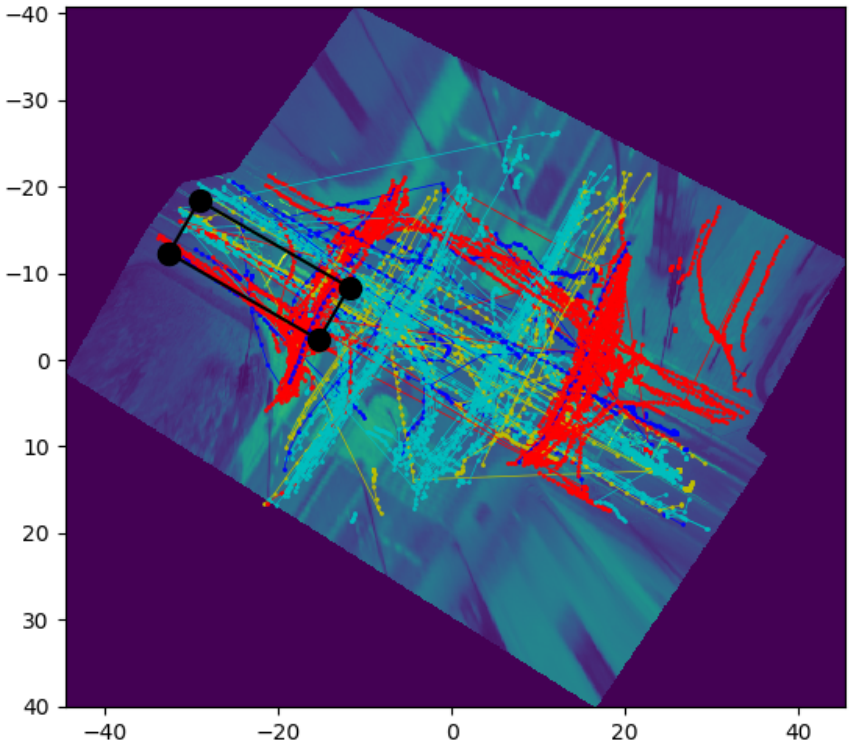

Figure 13: *The limited area for which all host objects analysed in this project are inside together with trajectories from traffic objects that all passes through the area. The colour of the trajectory denotes the type of the traffic object according to: red - pedestrian, blue - bicyclist, cyan - vehicle and yellow - heavy vehicle. Note that many of these random trajectories probably contains invalid fragments, as described and visualised in Figure 12.*

This limitation means that when computing the states only vehicles that are inside the rectangle will be taken as host objects. But attending object for computation of the occupation matrices do not necessarily need to be inside this area at the state computation time - but still they must be in the range of the 3x3 grid.

5.5.1 Testing different grid sizes

The size of the occupancy grid may have a large impact in the analysis when searching which scenarios that are found in this intersection. A grid that is too small will not capture relevant attending objects, while a grid that is too large will capture objects far away which does not really interact with the host object. In order to show and understand what consequences that choice will make we try mainly 3 different grid sizes (width x height): 7.5 m x 15 m, 15 m x 30 m and 30 m x 60 m. A visualisation of the 3 different sizes for an object within the area that is analysed can be found in Figure 14.

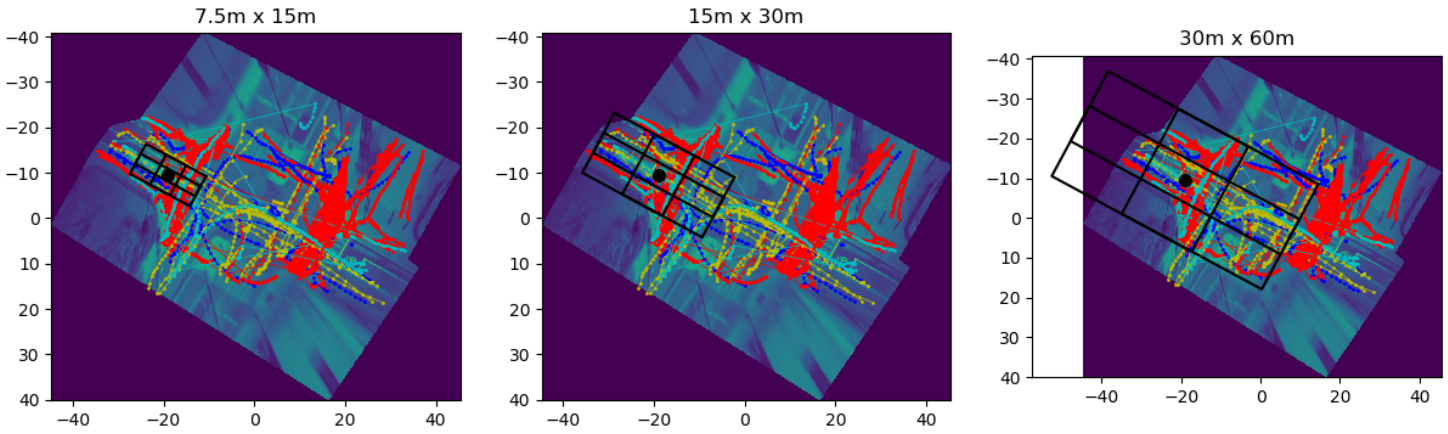

Figure 14: *Tested grid sizes for an example host object together with random trajectories from other traffic objects. Position of host object is marked by a black dot. Left panel shows a grid of size 7.5 m x 15 m, middle panel a size of 15 m x 30 m and the right panel a size of 30 m x 60 m*

5.5.2 Clustering sequences of states

To examine what scenarios that can be captured by transitions of the occupancy grids we have used clustering for sequences of consecutive state computations from host objects in the limited area. What length such sequence should be of is a big issue, and considered too large to be included in this work. For the analysis here we have used sequences of 10 consecutive state computations, and because there is 0.3 seconds between each state computation the length in time of the scenario is 3 seconds. We believe many scenarios might be both shorter and longer in time, but using this fixed time length simplifies the analysis of the actual states & transition model we want to investigate.

Because the interest in this analysis lies in the interaction between the host and attending objects, all hosts that have not any nearby attending objects have been removed before computing sequences. This depends on the size of the grid, why the number of analysed sequences may be greater when a larger grid is used.

When clustering such sequences, the vectors from each state computation are concatenated to a vector of length 360 (36 attributes for one state and 10 consecutive states). By using the representation of a sequence with this longer vector, the similarities (and dissimilarities) are explored by clustering samples of such.

5.6 Method for analysing states and transitions

One method for analysing states and transitions is plotting heatmap. Heatmap in Figure 13 shows the correlation among different variables of two consecutive states.

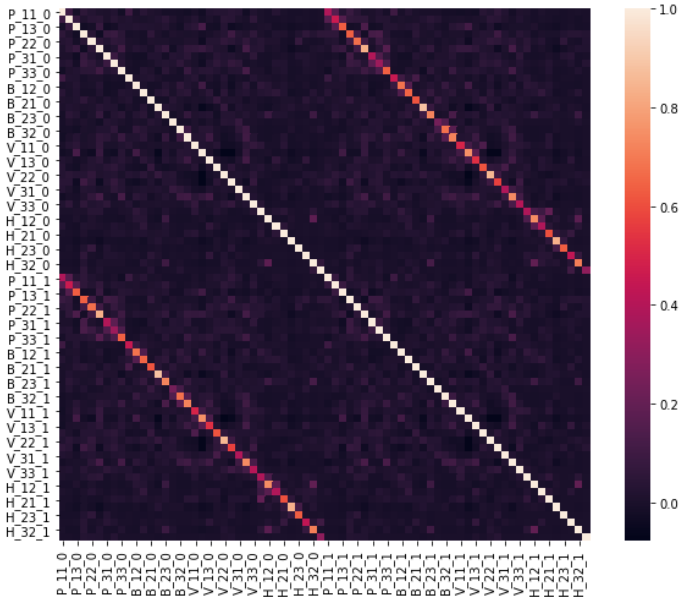

Figure 15: *Heatmap showing the correlation among consecutive states.*

6 Results

6.1 Characteristics of occupation matrices

In order to find out capabilities of the occupation matrices we have investigated computed states from vehicles in the limited area recorded from 2017-04-28 to 2017-05-20.

The total number of nonzero entries of the occupation matrix for all computed states in the grids increases as the size of the grid increases, which is shown in Figure 16. From this plot, we can see that the number of occupations increase most for pedestrians as the size of the grid increases. For the other object types the number increase, but with a much lower rate than for the pedestrians.

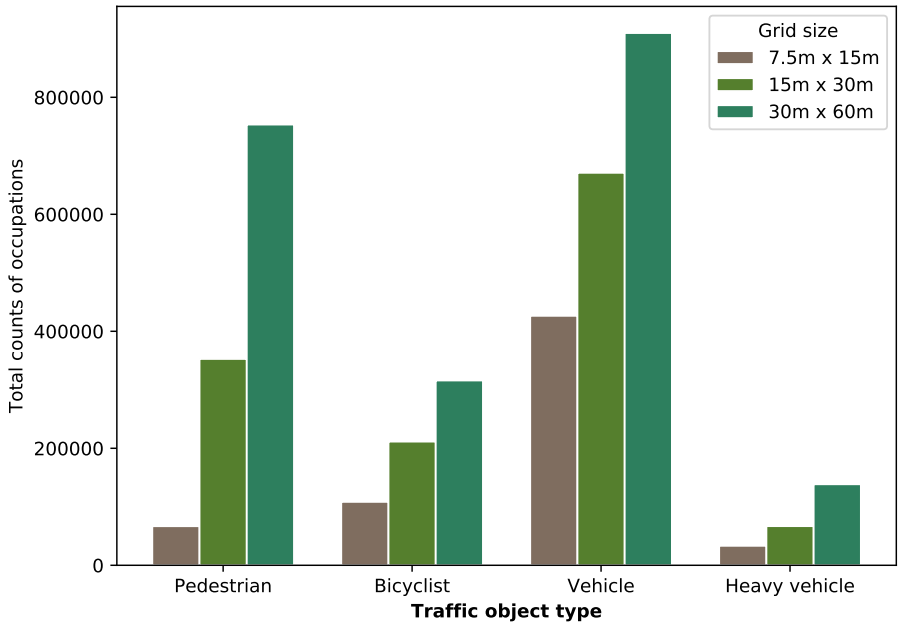

Figure 16: *The total number of nonzero entries of the occupation matrix for all computed states of the different traffic object types for the 3 different grid sizes.*

When investigating what the occupancy matrices looks like in a single state we see that the behaviour is very different depending on traffic object type the size of the grid, which is visualized in Figure 17. For the pedestrian matrix there is a pattern of pedestrians to be more likely to be to the right or at the front of the vehicle (for the two smaller grid sizes). In the bicyclists matrix the most common occupation, independent on grid size, is in the middle cell. This pattern turn out to appear also for vehicles and heavy vehicles. When only looking on the grid sizes, we see that for all traffic object types the ratio of occupation in cells behind the vehicle decreases as the grid size increases.

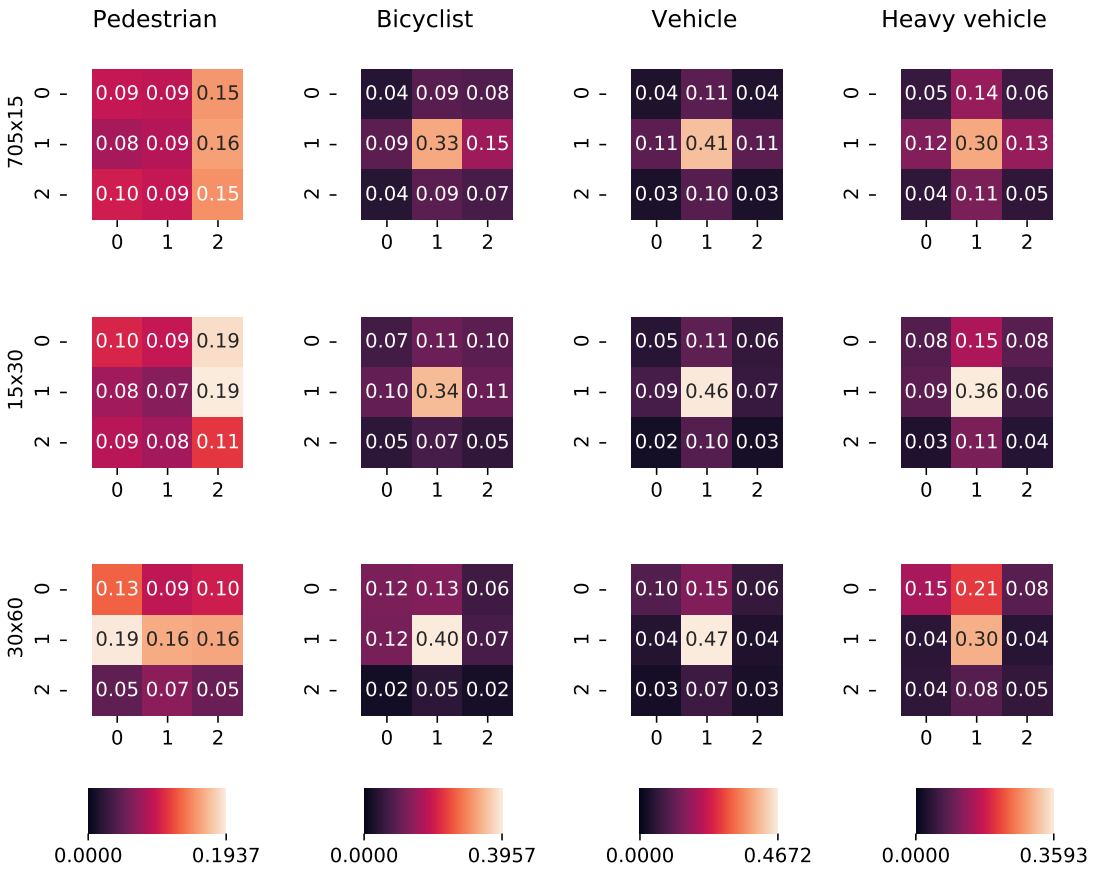

Figure 17: *Heatmaps of ratios of occupations of the grids for different sizes (rows) and traffic object types (columns). For each heatmap the value in a cell is computed by taking the sum of occupations in a particular cell of any state computation and divide by the sum of occupations in all cells (for the matrix of the given traffic object type and grid size). The color scale, visualised at the bottom, is the same for every plot of same traffic object type, e.g. by column.*

6.2 Clustering scenarios using occupation matrices

To find out what scenarios that appear in the extracted sequences clustering have been used. Mainly two clustering methods have been used for investigating this structure: DBSCAN and mini-batch K-means. DBSCAN fails to produce informative clustering, while K-means reveals some underlying structures.

6.2.1 DBSCAN

When using DBSCAN the results have been hard and not interesting to interpret, due to unsatisfactory cluster sizes. When setting ϵ to a small value (about 1-$\sqrt{7}$) about 10 different clusters appears, but all of them expect one have very few samples assigned to them. The rest is either assigned as outliers or to the large cluster where the majority of samples are found. When choosing a larger value of ϵ the smaller clusters dissappear and there are only outliers and one cluster with the vast majority of samples. Because of the roughness in the result, we choose to not present the results from DBSCAN in more detail.

6.2.2 k-means

In this section results from clustering using mini-batch K-means is presented. The number of clusters is set to 10 in all cases, because it reveals much information at the same time as it is quite simple to grasp. Using fewer or more clusters gives quite similar results, why this might be a reasonable starting point when analysing the model. In order to reduce redundant information only sequences from host objects that have nonzero occupation matrices in at least one state computation are used for clustering.

Clusters shows in some cases very interesting patterns that is close to the idea of a scenario. An example of three extracted cluster that clearly represents different scenarios can be seen in Figure 18 and Figure 19 where clustering has been performed using the middle size grid (15m x 30m). Cluster A represents pedestrians moving along the pavement to the right, and the host can pass the zebra crossing without stopping for pedestrians. Comparing with Cluster B, we can both see in the

heatmaps and in the visualisation that pedestrians moves a bit slower and some of them are passing, or have recently passed, the road on the zebra crossing. In the third cluster, C, there are many pedestrians passing on the zebra crossing, forcing the host to stop (which makes the trajectory)

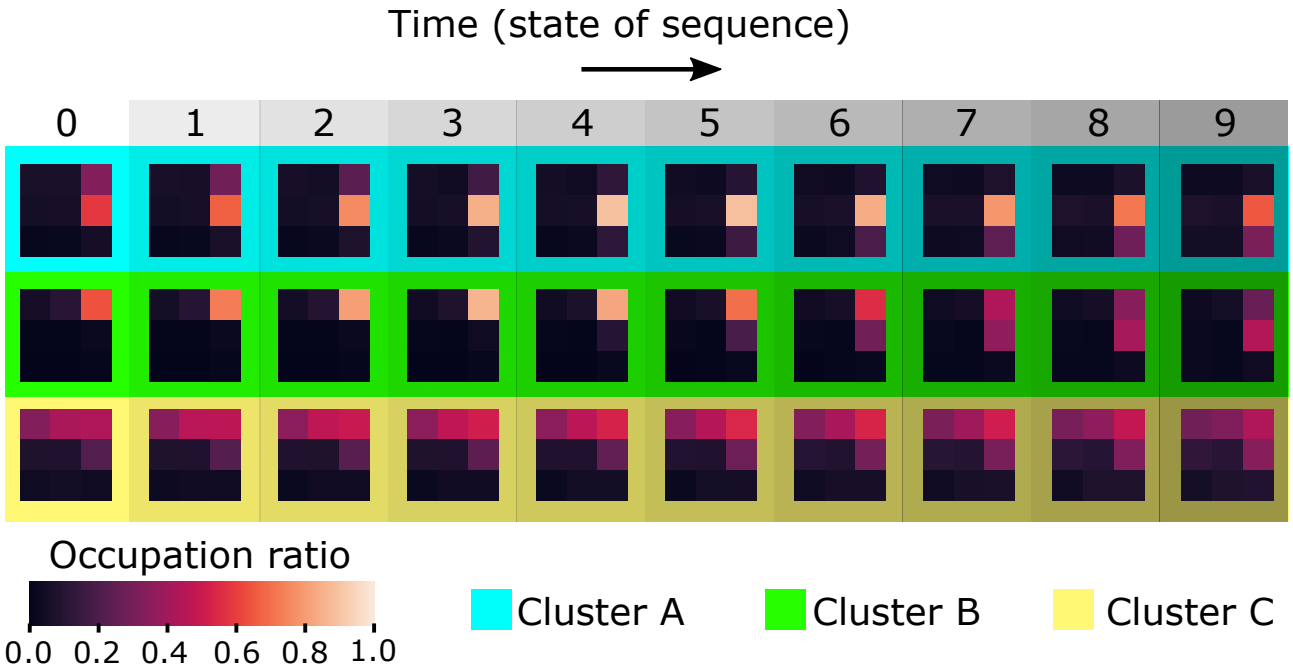

Figure 18: *Example of 3 clusters (A, B, C) representing different scenarios. Each cell shows a heatmap of the occupation ratios of the cells in the pedestrian occupation grid among sequences in the cluster given by the color of the row at the state given by the column. These clusters are computed in a separate run from the ones analysed later, why they might differ a bit from other results in this section.*

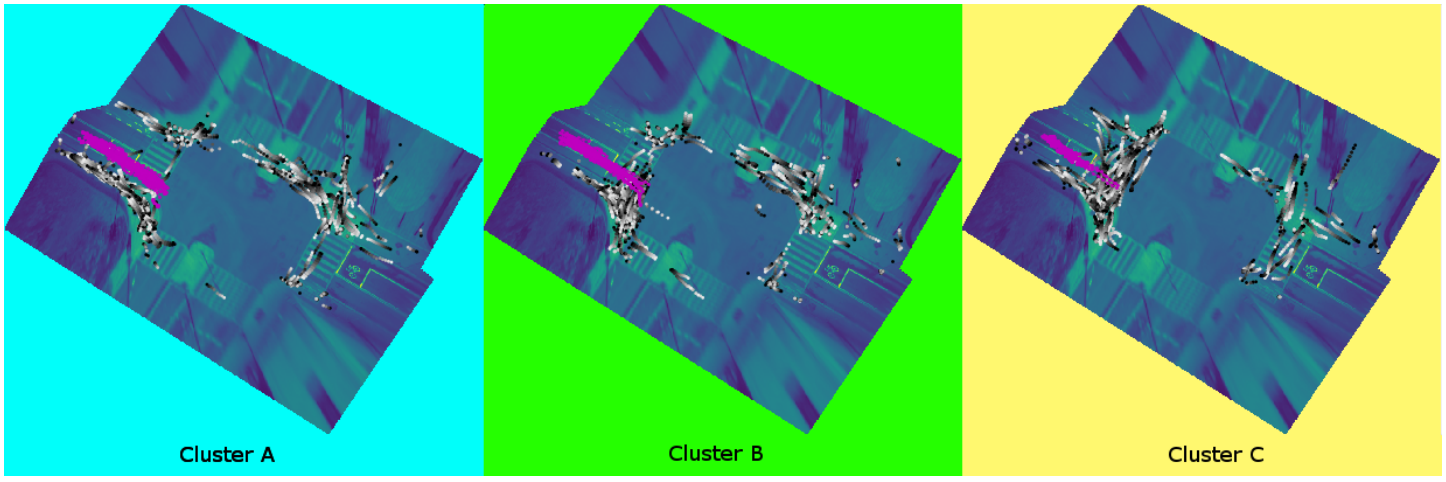

Figure 19: *A sample of 100 random trajectories from host vehicles (magenta) and pedestrians for the sequences (of 3 seconds) for each of the three clusters A, B and C (corresponding to the clusters in Figure 18). The color of the trajectory of the pedestrians depends on the order in sequence: the earlier in sequence the position is recorded the darker it is. For each recording of a (unique) pedestrian the color of its recorded positions are gradually lighter till the last position of the sequence (which is colored white).*

For a grid size of 7.5m x 15m sizes of clusters can be seen in Table 1. About half of the samples are assigned to cluster 0 and 16% to cluster 1 while the remaining clusters have a size of less than 10% of the samples. In Figure 20, where similar occupation patterns within clusters are visualised, cluster 0 and 1 are hardly visible. None of the clusters shows a similar occupation pattern for the pedestrian matrices. Cluster 6 is the only cluster that indicates a pattern for the heavy vehicles, with a significant peak at cell 2,2 in the grid. The rest of the clusters seems to mainly contain occupations of vehicles and bicyclists. The most common occupation in these clusters is the middle cell, while there are some clusters (e.g. 8) that have its most common occupation of each state in another cell.

For the bicyclists there are two clusters that shows some patterns: cluster 4 and 9. In cluster 4 the pattern is that there is a bicyclist in cell 2,2 during the whole sequence, while the pattern in cluster 9 is a bicyclist in cell 2,3, e.g. more to the right of the vehicle.

For the vehicles we see 4 clusters that shows a couple of distinct patterns: cluster 0, 2, 3, 7 and 8. In both cluster 0 and 3 the scenario seems to be a vehicle that occupies the middle cell during the whole sequence, while the most common pattern in cluster 2 is a vehicle coming closer to the host's front. Also cluster 7 and 8 contains scenarios when there is a vehicle of the front of the host, but further away (in cell 1,2). The difference between the clusters seems to be for 7 it is a vehicle that is close to the host and increases its distance, while 8 is the opposite.

Cluster id	Cluster size	Fractional size
0	20276	0.16
1	61174	0.49
2	8596	0.07
3	2205	0.02
4	7857	0.06
5	7249	0.06
6	1992	0.02
7	4186	0.03
8	6390	0.05
9	4375	0.04

Table 1: *Clusters and sizes using mini-batch K-means for grid of size 7.5m x 15m.*

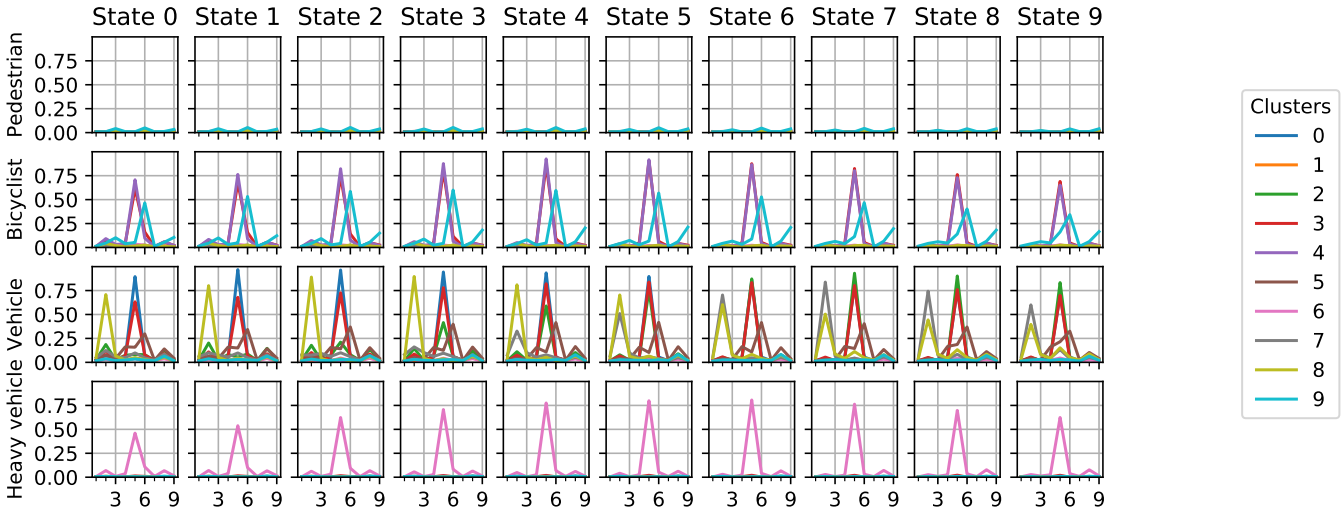

Figure 20: *Visualisation of K-means for grid of size 7.5m x 15m using 10 clusters. For each traffic object type and state computation of the sequence there is a subplot indicating occupations for the different clusters. The X-axis for each subplot is the occupation matrix flattened out (as in the state vector, e.g. 1-3 is first row of matrix, 4-6 second row and 7-9 last row) The Y-axis shows the ratio of sequences in the cluster that has occupied cells in the particular state computation.*

For a grid size of 15m x 30m sizes of clusters can be seen in Table 2. More than half of the samples are found in either cluster 0 or 1, while the the remaining clusters have a size smaller than 10% of all samples. In Figure 21 we see that some patterns among pedestrians appear, while there are no patterns of the heavy vehicles of any of the clusters. Comparing with 20, the changes between occupation ratios between states seems to be somewhat larger, while many ratios are similar when comparing neighbouring states.

Looking at the pedestrians for the middle size grid there are mainly two clusters that reveals two scenarios: cluster 3 (red) and 7 (gray). Cluster 3 seems to contain scenarios where pedestrians pass in the front and to the right of the vehicle. Interestingly, in this cluster we can also find a vehicle in the middle cell of the grid. On the other hand, cluster 7 only contains scenarios where pedestrians are to the right of the host object, and never to the left.

Cluster id	Cluster size	Fractional size
0	45547	0.25
1	61220	0.34
2	13293	0.07
3	4467	0.02
4	14845	0.08
5	2092	0.01
6	5094	0.03
7	11805	0.07
8	14655	0.08
9	7953	0.04

Table 2: *Clusters and sizes using mini-batch K-means for grid of size 15m x 30m.*

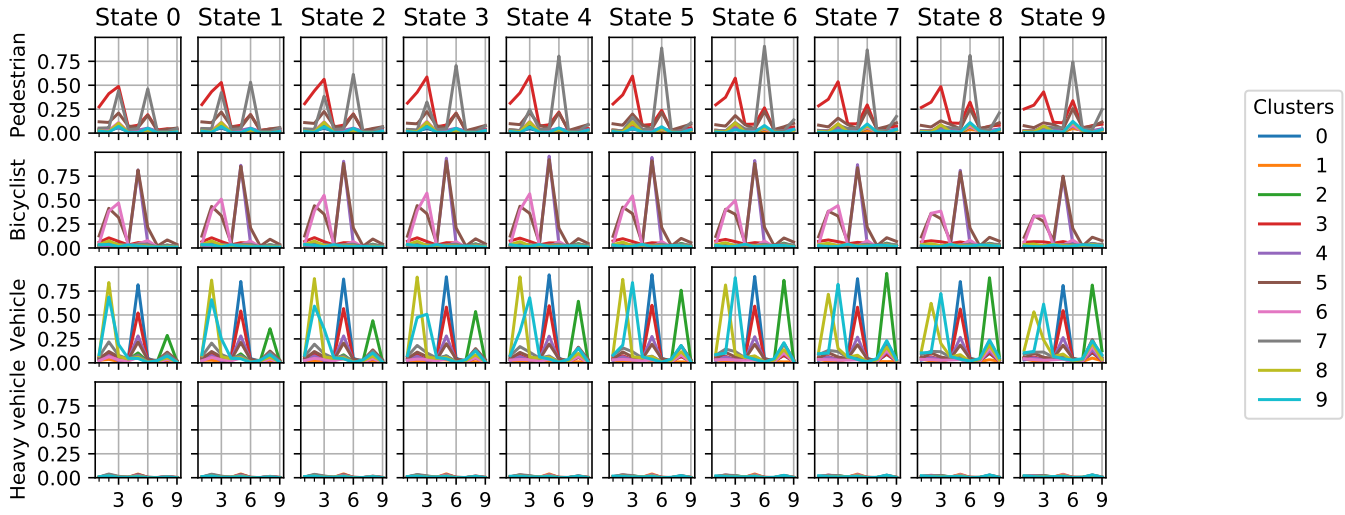

Figure 21: *Visualisation of K-means for grid of size 15m x 30m using 10 clusters. For more details, see Figure 20*

For a grid size of 30m x 60m sizes of clusters can be seen in Table 3. Also for this one there are two clusters (7 and 8) that contains much of the data, about 40%. Patterns among all the different types can be found in different clusters. Apart from vehicles, occupations between neighbouring states vary very little.

Cluster id	Cluster size	Fractional size
0	8625	0.05
1	17160	0.10
2	4872	0.03
3	13861	0.08
4	19914	0.11
5	13876	0.08
6	6835	0.04
7	49149	0.28
8	28897	0.17
9	10806	0.06

Table 3: *Clusters and sizes using mini-batch K-means for grid of size 30m x 60m.*

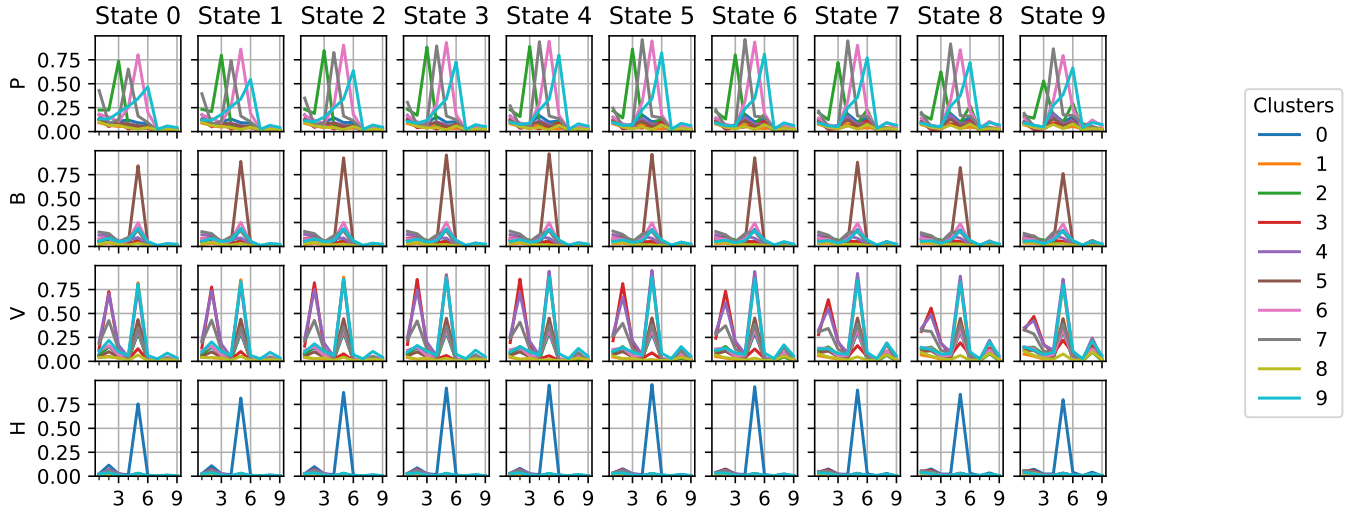

Figure 22: *Visualisation of K-means for grid of size 30m x 60m using 10 clusters. For more details, see Figure 20*

7 Discussion

We see some of the results as promising, but still there are many questions still unanswered and some remarks.

7.1 Occupation grid

Even though both model and analysis performed can be regarded as quite simple, we see some tendencies that this method is useful for extracting scenarios. The best example is from Figure 19 where you by eye can track the differences between the sequences of the different clusters.

Overall when analysing the performance of the clustering for the limited area (see Figure 15) we expect to find for instance bicyclists to the right of the host, pedestrians to the right and crossing the road ahead and oncoming vehicles to the left. We can find such tendencies among the clusters from k-means that such scenarios are found in the data. However, there are some certain problems with the model.

7.1.1 Grid size and structure

As expected we see in Figure 16 a larger grid results in more occupations for all object types.

When using the small grid (7.5m x 15m) it is quite obvious that it is too small in order to find scenarios involving pedestrians. First of all, we see in Figure 16 that the number of pedestrian occupations is very few compared with the other types and the relative change of pedestrian occupations to larger grids. Also in Figure 17 we see that the ratios for pedestrian occupations are quite similar in all cells for the smallest grid. The most distinct example of this is in Figure 20 where none of the extracted clusters shows similar patterns for pedestrians. For pedestrians the middle grid size (15m x 30 m) seems to be the most appropriate one of the three tried out here, because it is the only size with scenarios that have varying occupation ratios.

Even for the small grid the most common occupation of bicyclists is in the middle cell, as well as for vehicles and heavy vehicles (see Figure 17). But for both the small and medium grid there is a slight tendency that bicyclists are to the right (cell 2,3) of the vehicle. This is what is kind of expected according to what the road looks like, and it can also be verified by looking at cluster 9 in Figure 20 for the small grid and cluster 5 in Figure 21 for the medium grid. One difference of the extracted scenarios in the medium grid compared with the small is that bicyclists in the front row is extracted.

The largest grid is probably too large to describe the state of the host object for any object type, at least with this limited data. Much of the space of the cells behind the host is outside the area where objects are tracked, which can be seen in Figure 14, while the cells ahead of the host object is almost on a completely different part of the intersection. We also see in the heatmaps in Figure 17 that occupation mostly appears in the front and relatively rarely behind, which probably is a consequence of lack of data from behind. What is interesting is that especially for the vehicles in Figure 22 some patterns of movement are extracted in the clusters. Thus a larger grid gets more like a geolocated grid and might be useful for describing the scenario of the whole intersection. But, if that is the objective probably more cells than 9 should be needed to distinguish between different scenarios.

The definite answer on what grid size that should be used is that it is different for different object types due to their different behaviour. If restricted to choose only one size of the three tested, probably the middle sized grid is the most appropriate to describe the situation of the host object. An idea that we have not tried here is to let the grid size depend on the speed of the host object, which would make the size of the grid fixed in sense of time rather than actual distance.

7.1.2 Observed problems and suggested improvements

When have such binary data as in the occupation matrices, the parameter ϵ can only be used to restrict the total number of differences, e.g. when using Euclidean distances two sequences that differ with 5 positions in the state vector the distance between those will be $\sqrt{5}$. Because the occupation matrices are most often very sparse, the number of differences between two sequences might be very small in terms of distance even though they describe completely different scenarios. One change of the model which will allow smaller values of ϵ and probably better clustering results from DBSCAN is to have non-binary matrices where one traffic object can cause values larger than 0 for more than just the occupied cell. For instance, the value of a particular cell in the pedestrian matrix depends on the distance between the center of the cell and its closest pedestrian, instead of just setting 1 whenever there is a pedestrian inside the cell. One way would be to put a 2-dimensional Gaussian function, with some sensible variance, around each attending object to assign its values of the occupation matrix. That would make sequences where object occupies different, but neighbouring, cells more similar to each other, which should also be reasonable to account for noise between sequences of the same scenario.

That would be a potential improvement for bicyclists, vehicles and heavy vehicles, which almost independently on how small the grid is tends to be very close to the host vehicle as we can see in Figure 17. The fact that another traffic object is very close is of course interesting to know, but with the current model information about in what relative position such objects are present in is lost. We can see it in clustering results where for example cluster 0 in Figure 20, which is the second largest cluster, to a large extent only contains vehicles in cell 2,2. E.g. right now we are not able to distinguish between vehicles that are really close to the host in the front, in the back, to the right or to the left. Another approach to solve this problem than the previously suggested (even though they can be combined) can be another design of the occupation grid visualised in Figure 23. Instead of having a cell that encapsulates the host object, the center of mass of the host is the intersection of 4 cells (which are not rectangular in this design). This would allow distinguishing between nearby objects close to the front, back etc. even with keeping the occupation values to be binary.

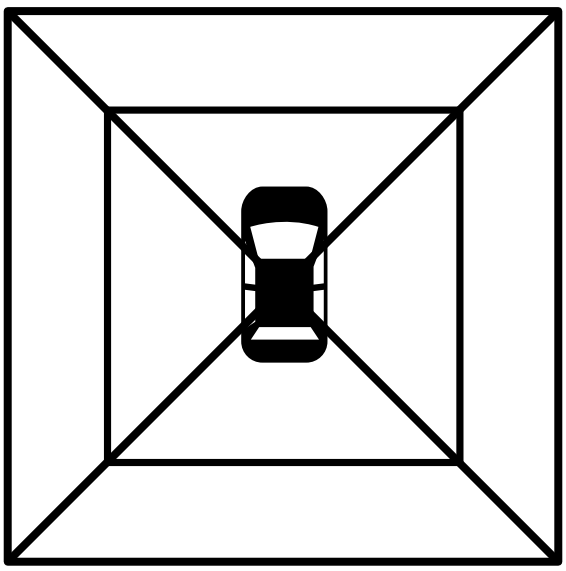

Figure 23: *Suggestion of another design of occupation grid. This design has not been tried out in this work.*

8 Concluding remarks

Occupation grid for each type of traffic object seems to be a promising approach for extracting scenarios, and its size should probably be determined individually for each kind of traffic object. However, both model and analysis can be performed with more sophisticated techniques improving results further.

Acknowledgments

We are thankful to Chalmers University of Technology and Volvo Cars for their support. We would also like to thank Viscando Traffic Systems AB for the dataset.

References

[1] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2nd ed. Springer, 2009. [Online]. Available: `http://www-stat.stanford.edu/~tibs/ElemStatLearn/` (visited on 04/15/2019).

[2] D. Sculley, "Web-scale k-means clustering", in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, pp. 1177–1178, ISBN: 9781605587998. DOI: 10.1145/1772690.1772862. [Online]. Available: https://doi.org/10.1145/1772690.1772862.

[3] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters 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*, ser. KDD'96, Portland, Oregon: AAAI Press, 1996, pp. 226–231.

A Appendix. Scripts for computing field data into states

A.1 ComputeStatesSequences.py

Main script for computing states

```
    import pandas as pd
from GridExtractor import GridExtractor
from LinearAlgebraUtils import *
from StateSequenceExtractor import StateSequenceExtractor
from DataUtils import *

### PARAMETERS ###

# Path or name to data file WITHOUT .csv
CSV_FILE = '2017-04-29'

# The separator char in csv file, for untouched fiels from Viscando this should not be changes
SEPARATOR = ';'

# Types/categories allowed to be hosts
host_types = [2]

# Create a grid extractor, define number of rows and columns
rows = 3
columns = 3
grid_extractor = GridExtractor(rows, columns)

# Set size of grid (7.5x15 corresponds to size of middle cell is size of a parking lot)
grid_width = 15
grid_height = 30

grids = {'15x30': (15, 30), '705x15': (7.5, 15), '30x60': (30, 60), '12x30': (12, 30)}

# Set time interval between state computations
computation_time_difference = .3

# Set maximum time interval between recordings (gaps larger than this results in invalid objects)
maximum_time_interval = .3

# Area (rotated rectangle) where hosts are allowed to be. Structure of matrix is:
#   | upper left, upper right |
#   | lower left, lower right |
```

```
area_1_corners = np.array([[[-32.649, -12.333], [-29.147, -18.394]], [[-15.33, -2.33], [-11.829, -8.391]]])
area_2_corners = np.array([[[14.34164079, -28.09968944], [25.07476708, -22.73312629]],
                           [[0.92523292, -1.26687371], [11.65835921, 4.09968944]]])
areas = {'area_2': RotatedRectangle(area_2_corners), 'area_1': RotatedRectangle(area_1_corners)}

### COMPUTATIONS ###

# Replace index duplicates with new indices and save new csv file
# Name of new file is {CSV_FILE}_unique_indices.csv (same directory as CSV_FILE)
#replace_index_duplicates(CSV_FILE, sep=SEPARATOR)

# Sort data by recording time and save new csv file
# Name of new file is {CSV_FILE}_unique_indices_by_time.csv (same directory as CSV_FILE)
#sort_by_time('{}_unique_indices'.format(CSV_FILE))

data_sorted_by_object_id = pd.read_csv('{}_unique_indices.csv'.format(CSV_FILE))
data_sorted_by_object_id.Time = pd.to_datetime(data_sorted_by_object_id.Time)

data_sorted_by_time = pd.read_csv('{}_unique_indices_by_time.csv'.format(CSV_FILE))
data_sorted_by_time.Time = pd.to_datetime(data_sorted_by_time.Time)

for grid, dims in grids.items():
    for area, rectangle in areas.items():
        states_extractor = StateSequenceExtractor(computation_time_difference, maximum_time_interval, host_ty
                                                  grid_extractor, rectangle)
        states_extractor.compute_state_sequence(data_sorted_by_object_id, data_sorted_by_time, dims[0], dims

        # Name of output file
        output = CSV_FILE + '_{}_{}.csv'.format(area, grid)
        states_extractor.states_sequences_df.to_csv(output, index=False)
```

A.2 Other necessary scripts and classes

Here follows a bunch of classes used from ComputeStatesSequences.py.

StateSequneceExtractor.py:

```
from GridExtractor import GridExtractor
from LinearAlgebraUtils import RotatedRectangle
from TrafficObjectTracker import TrafficObjectTracker
import numpy as np
import pandas as pd
from datetime import *
from StateComputingUtils import *


def get_df_cols(attending_types=['P', 'B', 'V', 'H'], occ_mat_rows=3, occ_mat_cols=3):
    cols = ['Time id', 'Object id']

    for object_type in attending_types:
        for i in range(occ_mat_rows):
            for j in range(occ_mat_cols):
                cols.append('{}_{}{}'.format(object_type, i + 1, j + 1))

    cols.extend(['Speed', 'Acceleration', 'Angular speed'])

    return cols


class StateSequenceExtractor:
    def __init__(self, delta_time_seconds: float, max_time_difference_seconds: float, host_types: list,
```

```
             grid_extractor: GridExtractor, limited_area: RotatedRectangle = None):
    """

    Parameters
    ----------
    delta_time_seconds: Time between each computation of states in seconds
    max_time_difference_seconds: Maximum allowed time difference between two recordings for one object.
    """
    self.limited_area = limited_area
    self.grid_extractor = grid_extractor
    self.host_types = host_types
    self.max_time_difference = timedelta(seconds=max_time_difference_seconds)
    self.delta_time = timedelta(seconds=delta_time_seconds)

    self.attendants = set()
    self.hosts = set()

    self.states_sequences_mat = None
    self.states_sequences_df = pd.DataFrame(columns=get_df_cols())
    self.append_index = 0

    self.obj_trackers = {}

    self.time_id = -1

    self.valid_attendants = True

    self.obj_ids_ordered = None
    self.n_rows = 0

def update_state_time(self, previous_state_time, index):
    if index == self.n_rows - 1:
        self.time_id += 1
        return previous_state_time + self.delta_time

    object_id = self.obj_ids_ordered[index]
    next_recording = self.obj_trackers[object_id].current().Time
    state_time = previous_state_time

    while next_recording >= state_time:
        state_time += self.delta_time
        self.time_id += 1

    return state_time

def init_obj_trackers(self, df_by_object: pd.DataFrame):
    self.obj_trackers = {}
    start = 0
    obj_id = df_by_object.iloc[0].ID
    for index, row in df_by_object.iterrows():
        if row.ID != obj_id:
            object_df = df_by_object.iloc[start:index]
            self.obj_trackers[obj_id] = TrafficObjectTracker(object_df, obj_id, self.max_time_difference
                                                             self.delta_time.total_seconds(), len(get_df_
                                                             limited_area=self.limited_area)
            print('Init traffic object trackers: {}%'.format(int(100 * index / self.n_rows)))

            start = index
            obj_id = row.ID

    object_df = df_by_object.iloc[start:]
```

```
    self.obj_trackers[obj_id] = TrafficObjectTracker(object_df, obj_id, self.max_time_difference,
                                                     self.delta_time.total_seconds(), len(get_df_cols()),
                                                     limited_area=self.limited_area)

def iter_records(self, state_time, index):
    if index == self.n_rows - 1:
        return index

    obj_id = self.obj_ids_ordered[index]
    tracker = self.obj_trackers[obj_id]

    while tracker.current().Time <= state_time:
        if tracker.pointer == 0:
            self.attendants.add(obj_id)

        if tracker.pointer == 2 and tracker.current().Type in self.host_types:
            self.hosts.add(obj_id)

        if not tracker.reached_end():
            tracker.inc_pointer()

        if index == self.n_rows - 1:
            return index

        index += 1
        obj_id = self.obj_ids_ordered[index]
        tracker = self.obj_trackers[obj_id]

    return index

def extract_attendant_positions(self, state_time):
    positions = {0: [], 1: [], 2: [], 3: []}
    dead_objects = []
    invalids = []
    for attendant in self.attendants:
        tracker = self.obj_trackers[attendant]
        if tracker.is_valid_attendant(state_time):
            row = tracker.current()
            positions[row.Type].append(np.array([row.X, row.Y]))

        else:
            if tracker.reached_end():
                dead_objects.append(attendant)
            else:
                row = tracker.current()
                invalids.append(np.array([row.X, row.Y]))

    return positions, invalids, dead_objects

def remove_dead_objs(self, dead_objs):
    for obj_id in dead_objs:
        tracker = self.obj_trackers[obj_id]
        self.attendants.remove(obj_id)
        if obj_id in self.hosts:
            self.hosts.remove(obj_id)
            host_sequence_data = tracker.get_sequence_data()
            if host_sequence_data.shape[0] > 0:
                self.append_host(host_sequence_data)

        del self.obj_trackers[obj_id]
```

```
    def append_host(self, host_data: np.ndarray):
        if self.append_index + host_data.shape[0] >= self.states_sequences_mat.shape[0]:
            self.states_sequences_mat = np.append(self.states_sequences_mat,
                                                  np.zeros((self.n_rows, len(get_df_cols()))), axis=0)

        self.states_sequences_mat[self.append_index: self.append_index + host_data.shape[0], :] = host_data
        self.append_index += host_data.shape[0]

    def compute_state_sequence(self, df_by_object: pd.DataFrame, df_chronologically: pd.DataFrame, grid_width
                               grid_height):
        first_time = df_chronologically.iloc[0].Time
        last_time = df_chronologically.iloc[-1].Time + 2 * self.max_time_difference

        self.n_rows = df_chronologically.shape[0]
        self.obj_ids_ordered = df_chronologically.ID
        del df_chronologically

        self.init_obj_trackers(df_by_object)
        del df_by_object

        self.states_sequences_mat = np.zeros((self.n_rows, len(get_df_cols())))

        state_time = first_time
        index = 0
        self.time_id += 1

        while state_time < last_time:
            index = self.iter_records(state_time, index)

            print('Compute states: {}%'.format(int(100 * index / self.n_rows)))

            attendants_dict, invalids, dead_objs = self.extract_attendant_positions(state_time)

            self.remove_dead_objs(dead_objs)

            for host in self.hosts:
                tracker = self.obj_trackers[host]
                if tracker.is_valid_host():
                    previous_position, current_position = tracker.get_movement()
                    grid = self.grid_extractor.compute_grid(previous_position, current_position, grid_width,
                                                            grid_height)

                    no_invalids_in_grid = True
                    for invalid in invalids:
                        if grid.is_inside(invalid):
                            no_invalids_in_grid = False
                            break

                    if not no_invalids_in_grid:
                        continue

                    tracker.append_state(self.time_id, attendants_dict, grid)

            state_time = self.update_state_time(state_time, index)

        self.states_sequences_df = pd.DataFrame(data=self.states_sequences_mat[:self.append_index],
                                                columns=get_df_cols())
        print('Finished')
```

TrafficObjectTracker.py:

```
import math
```

```
import pandas as pd
import datetime

from LinearAlgebraUtils import *
from StateComputingUtils import compute_occupancy_matrices


class TrafficObjectTracker:
    def __init__(self,
                 df: pd.DataFrame,
                 object_id: int,
                 max_time_gap: datetime.timedelta,
                 delta_time_seconds: float,
                 n_columns: int,
                 pointer=0,
                 limited_area: RotatedRectangle = None):

        self.object_id = object_id
        self.limited_area = limited_area
        self.max_time_gap = max_time_gap
        self.pointer = pointer
        self.df = df.sort_values(by='Time')
        self.was_valid = True
        self.is_valid = True
        self.n_rows = self.df.shape[0]

        self.state_sequence_data = np.zeros((self.get_max_rows(delta_time_seconds), n_columns))
        self.state_pointer = 0

    def is_valid_attendant(self, state_time):
        return state_time - self.df.iloc[self.pointer].Time < self.max_time_gap

    def is_valid_host(self):
        if self.pointer > 1:
            current_position = self.current()[['X', 'Y']].values
            within_limited_area = self.limited_area is None or self.limited_area.is_inside(current_position)
            valid_time_gap = self.df.iloc[self.pointer].Time - self.df.iloc[
                self.pointer - 2].Time < 2 * self.max_time_gap
            return within_limited_area and valid_time_gap
        return False

    def get_speed(self):
        return self.df.iloc[self.pointer].Speed

    def get_acceleration(self):
        delta_speed = self.df.iloc[self.pointer].Speed - self.df.iloc[self.pointer - 1].Speed
        delta_time = (self.df.iloc[self.pointer].Time - self.df.iloc[self.pointer - 1].Time).total_seconds()

        return delta_speed / delta_time

    def get_movement(self):
        previous_position = self.df.iloc[self.pointer - 1][['X', 'Y']].values
        current_position = self.df.iloc[self.pointer][['X', 'Y']].values

        return previous_position, current_position

    def get_delta_angle(self):
        delta_time = (self.df.iloc[self.pointer].Time - self.df.iloc[self.pointer - 2].Time).total_seconds()

        previous_previous_position = self.df.iloc[self.pointer - 2][['X', 'Y']].values
        previous_position = self.df.iloc[self.pointer - 1][['X', 'Y']].values
```

```
        current_position = self.df.iloc[self.pointer][['X', 'Y']].values
        delta_theta = compute_delta_theta(previous_previous_position, previous_position, current_position)

        return delta_theta / delta_time

    def reached_end(self):
        return self.pointer + 1 == self.n_rows

    def inc_pointer(self):
        self.pointer += 1

    def current(self):
        return self.df.iloc[self.pointer]

    def append_state(self, time_id, attendants_dict, grid: Grid):

        state_vector = np.zeros(2 + 4 * 9 + 3)
        state_vector[0] = time_id
        state_vector[1] = self.object_id

        occupancy_matrices = compute_occupancy_matrices(attendants_dict, grid, self.current()[['X', 'Y']].val

        for object_type in [0, 1, 2, 3]:
            np.put(state_vector,
                   list(range(2 + object_type * 9, 2 + (object_type + 1) * 9)),
                   occupancy_matrices[object_type].flatten())

            state_vector[2 + 4 * 9] = self.get_speed()
            state_vector[2 + 4 * 9 + 1] = self.get_acceleration()
            state_vector[2 + 4 * 9 + 2] = self.get_delta_angle()

        self.state_sequence_data[self.state_pointer] = state_vector
        self.state_pointer += 1

    def get_sequence_data(self):
        return self.state_sequence_data[:self.state_pointer, :]

    def get_max_rows(self, time_difference):
        last_time = self.df.iloc[-1].Time + self.max_time_gap
        first_time = self.df.iloc[0].Time
        diff = last_time - first_time
        return math.ceil(diff.total_seconds() / time_difference)
```

DataUtils.py:

```
    import pandas as pd


def replace_index_duplicates(file_name, sep=','):
    df = pd.read_csv('{}.csv'.format(file_name), sep=sep)
    n_rows = df.shape[0]
    max_index = max(df.ID)

    previous_id = df.iloc[0].ID
    indices = {previous_id}
    change_index = None

    for index, row in df.iterrows():
        if row.ID != previous_id:
            if change_index is not None and row.ID == change_index:
                df.iat[index, 0] = previous_id
            elif row.ID in indices:
```

```
                change_index = row.ID
                max_index += 1
                df.iat[index, 0] = max_index
                previous_id = max_index
                indices.add(max_index)
            else:
                previous_id = row.ID
                indices.add(row.ID)
                change_index = None
            print('Replace index duplicates: {}%'.format(int(100 * index / n_rows)))

    df.to_csv('{}_unique_indices.csv'.format(file_name), index=False)


def sort_by_time(file_name, sep=','):
    print('Sort date frame by time')
    df = pd.read_csv('{}.csv'.format(file_name), sep=sep)
    df.Time = pd.to_datetime(df.Time)
    df = df.sort_values(by=['Time'])
    df.to_csv('{}_by_time.csv'.format(file_name), index=False)
```

GridExtractor.py:

```
import numpy as np
from LinearAlgebraUtils import Grid


class GridExtractor:

    def __init__(self, rows, cols):
        """
        Parameters
        ----------
        df: DataFrame of records from all traffic objects
        object_id: Object ID for the object for which this GridExtractor will compute grids for
        grid_width
        grid_height
        index_memory: Index to start search for rows within allowed time interval for a state
        """
        self.cols = cols
        self.rows = rows

    def compute_grid(self, previous_position, current_position, width, height) -> Grid:
        """

        Returns
        -------
        A numpy array (2x2x2 matrix) of all the corners of the aligned grid. Structure of matrix is

                  column 0:    column 1:
        row 0: | [x_tl, y_tl], [x_tr, y_tr] |
        row 1: | [x_ll, y_ll], [x_lr, y_lr] |

        (x_tl = x coord for top left corner, y_lr= y coord for lower right corner)
        """
        # Compute the move direction and its orthogonal direction
        move = current_position - previous_position
        normed_move = move / np.linalg.norm(move)
        normed_orthogonal = np.array([-normed_move[1], normed_move[0]])

        # Compute positions of corners of the grid
        lower_left = current_position - .5 * height * normed_move - .5 * width * normed_orthogonal
```

```
        lower_right = current_position - .5 * height * normed_move + .5 * width * normed_orthogonal
        upper_left = current_position + .5 * height * normed_move - .5 * width * normed_orthogonal
        upper_right = current_position + .5 * height * normed_move + .5 * width * normed_orthogonal

        corners = np.array([[upper_left, upper_right], [lower_left, lower_right]])

        return Grid(corners, self.rows, self.cols)
```

LinearAlgebraUtils:

```
import numpy as np


def compute_rotation(lower_left, upper_left):
    return np.arctan2(upper_left[0] - lower_left[0], upper_left[1] - lower_left[1])


def compute_delta_theta(p0, p1, p2):
    """

    Parameters
    ----------
    p0 - the oldest recording
    p1 - recording in middle
    p2 - the next recording

    Returns
    -------

    """
    angles = []
    for pa, pb in [(p1, p0), (p2, p1)]:
        x_diff = pa[0] - pb[0]
        y_diff = pa[1] - pb[1]
        angles.append(np.arctan2(y_diff, x_diff))

    return angles[1] - angles[0]


class RotatedRectangle:
    def __init__(self, corners):
        self.height = np.linalg.norm(corners[1, 0] - corners[0, 0])
        self.width = np.linalg.norm(corners[0, 1] - corners[0, 0])
        self.corners = corners
        self.rotation = compute_rotation(corners[1, 0], corners[0, 0])

    def rotate_point(self, point):
        p = point - self.corners[1, 0]
        angle = self.rotation
        x = p[0] * np.cos(angle) - p[1] * np.sin(angle)
        y = p[0] * np.sin(angle) + p[1] * np.cos(angle)

        return np.array([x, y])

    def is_inside(self, point: np.array):
        # rotate point (relative to lower left corner)
        rotated_point = self.rotate_point(point)
        return 0 < -rotated_point[0] < self.width and 0 < rotated_point[1] < self.height


class Grid(RotatedRectangle):
```

```
    def __init__(self, corners, rows, cols):
        super().__init__(corners)
        self.cols = cols
        self.rows = rows
        self.cell_height = self.height / rows
        self.cell_width = self.width / cols

    def compute_cell(self, point):
        rotated_point = self.rotate_point(point)

        row = self.rows - 1 - int(rotated_point[1] / self.cell_height)
        col = int(-rotated_point[0] / self.cell_width)

        return row, col
```

StateComputingUtils.py:

```
from datetime import datetime

import numpy as np
from LinearAlgebraUtils import Grid


#                  description of the grid variable names:
#                  grid_xy is a 4x4x2 matrix
#    __ __ __      grid_xy[0,0] = upper left corner coordinates
#   |__|__|__|     grid_xy[0,3] = upper right corner coord.
#   |__|__|__|     grid_xy[3,0] = lower left corner coord.
#   |__|__|__|     grid_xy[3,3] = lower right corner coord.
#                  grid_xy[0,1] = upper 2nd (from left) corner coord.
#


def compute_occupancy_matrices(positions, grid: Grid, host_position):
    occupancy_matrices = {k: np.zeros((3, 3)) for k in positions.keys()}
    for object_type in occupancy_matrices.keys():
        for attendant_position in positions[object_type]:
            if np.linalg.norm(host_position - attendant_position) > 1e-6 and grid.is_inside(attendant_positio
                row, col = grid.compute_cell(attendant_position)
                occupancy_matrices[object_type][row, col] = 1

    # returns a dictionary containing occupancy matrices
    return occupancy_matrices
```

A.3 Script for computing sequences of states

Computes sequnece vectors using csv-files with computed states.

```
import pandas as pd
import numpy as np
import glob


def compute_non_zero(file, seq_len):
    remove_empty_occ_mat(file)
    compute_sequence(file + '_nonzero', seq_len)


def remove_empty_occ_mat(file, cov_start=2):
    df = pd.read_csv(file + '.csv')
    cov_end = cov_start + 4 * 9
    last_object = df.iloc[0]['Object id']
```

```
    last_time = df.iloc[0]['Time id']
    non_empty = df.iloc[0, cov_start:cov_end].any()
    start = 0

    for index, row in df.iloc[1:].iterrows():
        r = row[cov_start:cov_end]
        if not (row['Object id'] == last_object and row['Time id'] == last_time + 1):
            if not non_empty:
                df = df.drop(list(range(start, index)), axis=0)
            start = index
            non_empty = r.any()

        if not non_empty:
            non_empty = r.any()
        last_object = row['Object id']
        last_time = row['Time id']

    df = df.iloc[:, :cov_end]
    df.to_csv(file + '_nonzero.csv', index=False)


def compute_sequence(file, sequence_length, cov_start=2):
    df = pd.read_csv(file + '.csv')
    columns = list(df.columns[:cov_start])
    no_cov_cols = len(columns)
    n_covariates = len(df.columns[cov_start:])
    for i in range(sequence_length):
        columns.extend([c + '_{}'.format(i) for c in df.columns[cov_start:]])
    n_columns = len(columns)

    sequence_data = np.empty((df.shape[0], n_columns))
    sequence_count = 0
    last_object = df.iloc[0]['Object id']
    last_time = df.iloc[0]['Time id']
    seq = np.zeros(n_columns)
    np.put(seq, list(range(0, no_cov_cols + n_covariates)), df.iloc[0].values)
    sequences = [seq]

    for index, row in df.iloc[1:].iterrows():
        if row['Object id'] == last_object and row['Time id'] == last_time + 1:
            for i in range(len(sequences)):
                indices = list(range(no_cov_cols + n_covariates * (i + 1),
                                     no_cov_cols + n_covariates * (i + 2)))
                np.put(sequences[i], indices, row[cov_start:].values)
                if i == sequence_length - 2:
                    sequence_data[sequence_count, :] = sequences[i]
                    sequence_count += 1
                    del sequences[-1]
        else:
            print(index)
            sequences = []

        new_seq = np.zeros(n_columns)
        np.put(new_seq, list(range(0, no_cov_cols + n_covariates)), row.values)
        sequences.insert(0, new_seq)

        last_object = row['Object id']
        last_time = row['Time id']

    sequence_data = sequence_data[:sequence_count, :]
    sequence_df = pd.DataFrame(data=sequence_data, columns=columns)
```

```
    sequence_df.to_csv('seq_{}_'.format(sequence_length) + file + '.csv', index=False)


SEQUENCE_LENGTHS = [10]

files = glob.glob("*.csv")
for file in files:
    for length in SEQUENCE_LENGTHS:
        compute_sequence(file[:-4], length)
        compute_non_zero(file[:-4], length)
```

A.4 Script for analysis of occupation matrices

Creating plots and perform clustering on sequences of states.

```
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.ticker import MultipleLocator
from sklearn.cluster import DBSCAN, KMeans, MiniBatchKMeans


def plot_cluster_occ_ratios(file):
    df = pd.read_csv(file)
    labels = df['label']
    df = df.iloc[:, 1:]

    types = ['P', 'B', 'V', 'H']
    type_names = ['Pedestrian', 'Bicyclist', 'Vehicle', 'Heavy vehicle']

    fig, ax = plt.subplots(len(types), 11, sharex=True, sharey=True, figsize=(10, 4))
    for l in np.unique(labels):
        cluster_df = df[labels == l]
        for i, t in enumerate(types):
            cols = [c for c in cluster_df.columns if c[0] == t]
            ax[i, 0].set_ylabel(type_names[i])
            for j in range(10):
                if i == 0:
                    ax[i, j].title.set_text('State {}'.format(j))
                state_cols = cols[j * 9:(j + 1) * 9]
                freq = cluster_df.loc[:, state_cols].sum() / cluster_df.shape[0]
                x = np.arange(1, freq.shape[0] + 1)
                if i == 1 and j == 9:
                    ax[i, j].plot(x, freq, label=l)
                else:
                    ax[i, j].plot(x, freq)
                ax[i, j].set_ylim([0, 1])
                ax[i, j].set_yticks(np.arange(0, 1, .25))
                ax[i, j].xaxis.set_major_locator(MultipleLocator(3))
                ax[i, j].xaxis.set_minor_locator(MultipleLocator(1))
                ax[i, j].grid(True)

        s = str(l) + ' & {}'.format(cluster_df.shape[0]) + ' & {0:.2f}'.format(
            cluster_df.shape[0] / df.shape[0]) + '\\\\ \\hline'
        print(s)
    for i, t in enumerate(types):
        ax[i, -1].axis('off')

    fig.legend(loc='center right', title='Clusters')
```

```
    plt.show()


def plot_cluster_heatmaps(df, labels, clusters, type):
    fig, ax = plt.subplots(len(clusters), 10, sharey=True, sharex=True)
    for i, c in enumerate(clusters):
        cluster_df = df[labels == c]
        cols = [c for c in cluster_df.columns if c[0] == type]
        ax[i, 0].set_ylabel('Cluster {}'.format(c))
        for j in range(10):
            if i == 0:
                ax[i, j].title.set_text('State {}'.format(j))
            state_cols = cols[j * 9:(j + 1) * 9]
            freq = cluster_df.loc[:, state_cols].sum() / cluster_df.shape[0]
            heatmap = np.array([freq[:3], freq[3:6], freq[6:9]])

            sns.heatmap(heatmap, vmin=0, vmax=1, ax=ax[i, j], annot=False, fmt='.2f', cbar=False, square=True
            ax[i, j].axis('off')
            ax[i, j].set_xlim([-.5, 3])
            ax[i, j].set_ylim([3, -.5])
    plt.axis('off')
    plt.show()


def compute_type_matrices(types, df: pd.DataFrame, max_prob):
    type_matrices = []
    for col_index, t in enumerate(types):
        type_matrix = np.zeros((3, 3))
        for i in range(3):
            for j in range(3):
                type_matrix[i, j] = df[t + '_{}{}'.format(i + 1, j + 1)].sum()
        print(t, np.sum(type_matrix))
        type_matrix = type_matrix / np.sum(type_matrix)

        max_prob[col_index] = max(max_prob[col_index], np.max(type_matrix))
        type_matrices.append(type_matrix)

    return type_matrices, max_prob


def plot_area_heat_maps():
    files = glob.glob("*.csv")
    types = ['P', 'B', 'V', 'H']
    type_names = ['Pedestrian', 'Bicyclist', 'Vehicle', 'Heavy vehicle']
    area_sizes = ['705x15', '15x30', '30x60']
    areas = ['area_1']  # , 'area_2']

    for area in areas:
        fig, axes = plt.subplots(len(area_sizes), len(types))
        # fig.suptitle(area.replace('_', ' ').capitalize(), va='top')
        cbar_axes = []
        for col_index in range(len(types)):
            margin = .12
            width = .12
            left = margin + (width + margin) * col_index
            cbar_axes.append(fig.add_axes([left, .05, width, .05]))

        for ax, col in zip(axes[0], type_names):
            ax.set_title(col, va='bottom')

        max_prob = [0 for t in types]
```

```
        matrices = []
        for row_index, area_size in enumerate(area_sizes):
            area_files = [f for f in files if 'seq' not in f and area in f and area_size in f and 'nonzero' i
            area_df = pd.read_csv(area_files[0])
            for f in area_files[1:]:
                area_df = area_df.append(pd.read_csv(f))
            print(area_size)
            type_matrices, max_prob = compute_type_matrices(types, area_df, max_prob)
            matrices.append(type_matrices)

        for row_index, area_size in enumerate(area_sizes):
            for col_index, t in enumerate(types):
                sns.heatmap(matrices[row_index][col_index], ax=axes[row_index, col_index], square=True, vmin=
                            vmax=max_prob[col_index],
                            cbar=row_index,
                            cbar_ax=None if not row_index else cbar_axes[col_index],
                            cbar_kws={"orientation": "horizontal", "ticks": [0, max_prob[col_index]]},
                            annot=True,
                            fmt='.2f')
                axes[row_index, col_index].set_xlim([-.5, 3])
                axes[row_index, col_index].set_ylim([3, -.5])
            axes[row_index, 0].set_ylabel(area_size)

        fig.tight_layout(rect=[0, .1, 1, 1])
        plt.show()


def occupation_histograms():
    # set width of bar
    barWidth = 0.25

    small = [66855, 108434, 426479, 33334]
    medium = [352758, 211464, 670955, 66963]
    large = [753419, 315937, 909885, 138532]

    r1 = np.arange(len(small))
    r2 = [x + barWidth for x in r1]
    r3 = [x + barWidth for x in r2]

    plt.bar(r1, small, color='#7f6d5f', width=barWidth, edgecolor='white', label='7.5m x 15m')
    plt.bar(r2, medium, color='#557f2d', width=barWidth, edgecolor='white', label='15m x 30m')
    plt.bar(r3, large, color='#2d7f5e', width=barWidth, edgecolor='white', label='30m x 60m')

    # Add xticks on the middle of the group bars
    plt.xlabel('Traffic object type', fontweight='bold')
    plt.xticks([r + barWidth for r in range(len(small))], ['Pedestrian', 'Bicyclist', 'Vehicle', 'Heavy vehic
    plt.ylabel('Total counts of occupations')

    # Create legend & Show graphic
    plt.legend(title='Grid size')
    plt.show()


def perform_k_means(area_size):
    files = glob.glob("*.csv")
    area = 'area_1'
    area_files = [f for f in files if 'seq' in f and area in f and area_size in f and 'nonzero' in f]
    df = pd.read_csv(area_files[0])
    df.insert(0, 'date', area_files[0][7:17])
    for f in area_files[1:]:
        area_df = pd.read_csv(f)
```

```
        area_df.insert(0, 'date', f[7:17])
        df = df.append(area_df)

    cluster_alg = MiniBatchKMeans(n_clusters=10, batch_size=10000)
    cluster_alg.fit(df.iloc[:, 3:])

    df.insert(0, 'label', cluster_alg.labels_)
    df.to_csv('clusters_{}.csv'.format(area_size), index=False)
```

B Appendix. Script for computing fixed time interval

B.1 histogram.py

Script for plotting histogram

```
#%%
try:
    from IPython import get_ipython
    get_ipython().magic('clear')
    get_ipython().magic('reset -f')
except:
    pass
#%% Load data
import pandas as pd

dataframe = pd.read_csv(r'C:\Users\Acer\Desktop\Project course\Oslo_Kristian IV gate - Fredriksgate\CSV\2017-
df = dataframe
#%% Compute time differences
import numpy as np
from dateutil import parser

id = 50000
result_array = np.array([])
for i in range(0, id):
    if (df.iloc[i]['ID'] == df.iloc[i+1]['ID']):
        date1 = parser.parse(df.iloc[i+1]['Time'])
        date2 = parser.parse(df.iloc[i]['Time'])
        diff = date1-date2
        result_array = np.append(result_array, diff.total_seconds())
list = result_array.tolist()
print(list)
len(list)
#%% Python program to find N largest element from given list of integers
def Nmaxelements(list1, N):
    final_list = []

    for i in range(0, N):
        max1 = 0

        for j in range(len(list1)):
            if list1[j] > max1:
                max1 = list1[j];

        list1.remove(max1);
        final_list.append(max1)

    print(final_list)

# Driver code
list1 = result_array.tolist()
N = 1300
```

```
# Calling the function
Nmaxelements(list1, N)
#%% Histogram plot
removed_list = [a for a in list if (a >= 0)if (a <= 1)]
len(removed_list)

import matplotlib.pyplot as plt

num_bins = 100
fig = plt.hist(removed_list, num_bins, facecolor='blue', alpha=0.5)
plt.savefig(r'C:\Users\Acer\Documents\GitHub\Volvo-cars-project\devosmita\fig.png')
```

C Appendix. Script for plotting heatmap for analysing states and transitions

C.1 heatmap.py

Script for plotting heatmap

```
#%%
try:
    from IPython import get_ipython
    get_ipython().magic('clear')
    get_ipython().magic('reset -f')
except:
    pass
#%%Step 1: Import the required libraries
import pandas as pd
import matplotlib.pyplot as plt
#%%Step 2: Load data
df = pd.read_csv(r'C:\Users\Acer\Documents\GitHub\Volvo-cars-project\devosmita\seq_10_2017-04-29_area_1_12x30
print(df)
#%% Plot heatmap
import seaborn as sns

X1 = df[['P_11_0', 'P_12_0', 'P_13_0', 'P_21_0', 'P_22_0', 'P_23_0', 'P_31_0', 'P_32_0', 'P_33_0', 'B_11_0',
         'P_11_1', 'P_12_1', 'P_13_1', 'P_21_1', 'P_22_1', 'P_23_1', 'P_31_1', 'P_32_1', 'P_33_1', 'B_11_1',

cor = X1.corr() #Calculate the correlation of the above variables
plt.figure(figsize = (10,8))
sns.heatmap(cor, square=True)
```

D Appendix. Script for computing angle

D.1 AngleExtractor.py

Script for computing angle

```
from math import atan2,degrees

def ComputeAngle(x1, y1, x2, y2):
    xDiff = x2 - x1
    yDiff = y2 - y1
    return degrees(atan2(yDiff, xDiff))
```