TECHNISCHE UNIVERSITÄT
ILMENAU
Department of Computer Science and Automation
Systems and Software Engineering Group

Master Thesis

Identification of Trajectory Anomalies in Uncertain Spatiotemporal Data

Registration Date: 1 April 2020
Submission Date: 30 November 2020

Supervisor: Prof. Dr.-Ing. Kai-Uwe Sattler (TU Ilmenau)
Supervisor: Prof. Igor Anikin (KNRTU-KAI)

Submitted by: Mardanova Aigul
Matriculation Number 62106
aigul.mardanova@tu-ilmenau.de

Abstract

Nowadays spatiotemporal data analytics plays an important role, and this work is intended to solve the task of analyzing spatiotemporal data, representing vehicle trajectories, to perform frequent trajectory patterns mining and detection of trajectory anomalies with the consideration of uncertainty of data. The uncertainty, arising from the use of data from CCTV cameras, can lead to a loss in the quality of trajectory clustering results, their further classification and anomaly detection.

In the course of this work, the main existing approaches to analyze trajectories obtained from CCTV cameras, and, in particular, to solve the problem of detecting anomalies were considered, the reasons and features of the uncertainty of the initial data were inspected. As a result, an approach to solve the problem was proposed, the possibility of improving the accuracy of the results was investigated, and a method to minimize the effect of data uncertainty by taking into consideration the location of a moving vehicle in relation to the camera was developed. To implement the proposed approach, conduct assessment tests and visualize the results, a framework was developed.

Key words: spatiotemporal trajectory data, spatiotemporal data uncertainty, trajectories clustering, trajectory anomalies detection.

Contents

Abbreviations

CCTV	Closed-Circuit TeleVision
GIS	Geographic Information System
ITS	Intellectual Transport System
ST	Spatio-Temporal (data)
TVS	Traffic Video Surveillance
GPS	Global Positioning System
DTW	Dynamic Time Warping
LCSS	Longest Common SubSequence
FARS	Fatality Analysis Reporting System
ID	Identificator
SVM	Support Vector Machine
HMM	Hidden Markov Model
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DI	Dunn's Index
TS	Trajectory Simplification
RDP	Ramer-Douglas-Peucker (algorithm)
API	Application Programming Interface
AWT	Abstract Window Toolkit
GUI	Graphical User Interface
TTS	Total Sum of Squares
SSE	Sum of Squares due to Error

Notations

$\boldsymbol{\tau}$, $\boldsymbol{T}$	Trajectory
$\boldsymbol{TP}$	Trajectory Point
$\boldsymbol{C}$	Cluster
$\boldsymbol{CM}$	Cluster Model
$\boldsymbol{t}$	Time parameter
$\boldsymbol{x}$	X-coordinate
$\boldsymbol{y}$	Y-coordinate
$\boldsymbol{v_{avg}}$	Average speed
$\boldsymbol{d_{ij}}$	Euclidean distance between trajectories T_i, T_j
$\boldsymbol{dist(a_i^k, b_j^k)}$	Euclidean distance between points a_i^k, b_j^k
$\boldsymbol{R^2}$	Coefficient of Determination R-squared

1 Introduction

Nowadays spatiotemporal (ST) data analytics plays an important role in different applications, based on Geographic Information Systems (GIS). Recent advances in GIS and, in particular, in GIS technologies and infrastructure have made cities smarter. And Intellectual Transport Systems (ITS) with urban traffic analysis are one of the most attractive applications in a smart city [1]. Intelligence surveillance in smart cities has rapidly progressed in last decade [2]. More and more roads and public areas are getting equipped with monitoring video cameras, amount of publicly available video data increases further [3]. Automatic analysis in Traffic Video Surveillance (TVS) receives increasingly more attention [4].

Nowadays there are many tasks and applications of urban traffic analysis and, according to [2], tracking vehicles behavior using image processing of videos is one of the promising approaches. One of the main research approaches in urban traffic analysis, which works with data from monitoring video cameras, is mining frequent trajectory patterns from the ST data representing a traffic flow, because extracted trajectories can be afterwards applied to automatic visual surveillance, traffic management, suspicious activity detection, etc. [5][6]. Another important sub-category of traffic analysis, which has become a commended task in many applications in smart cities, is an identification of trajectory anomalies [2]. Anomaly is traditionally described as a data instance that remarkably deviates from the majority of data instances in a data set [7]. In TVS domain an anomalous activity refers to events violating the common rules [4]. Such unusual traffic patterns, which do not conform to expected behavior, reflect abnormal traffic streams on road networks and thus provide useful, important and valuable information [2]. For instance, when a traffic incident or jam happens, traffic flow changes suddenly, and this will be reflected by deviations from the normative activity patterns. That means that recognizing outliers can be useful in detecting traffic incidents. However, in the context of huge amounts of data to be processed, or information overload in other words, manual solutions are infeasible nowadays due to high complexity and high time consumption, and researchers look for automatic or semi-automatic intelligent methodologies to solve these tasks to minimize the required involvement of the human operator [8].

As stated in recent researches in the field of traffic data analysis, it is significant in many applications, including ITS, to take into consideration uncertainty of data. The reasons of data uncertainty can be imprecisions in measurements and inexactitude of observations. In case of acquiring trajectory data from video enforcement cameras data uncertainty can be caused by limitations of used devices or lost location [9].

1.1 Problem Statement

As it was mentioned above, ST data analytics plays an important role in everyday life, and the process of extracting useful information from ST data is one of the most significant challenges

in traffic data mining. Since ST trajectory data is multi-dimensional and spatiotemporally related, traditional data mining approaches, proposed for static, single and independent data, are inefficient and inappropriate in that case [10].

The main objective of the work in this thesis is to propose an approach configured to process the uncertain ST trajectory data to perform frequent trajectory patterns acquisition and detecting the anomalies, and perform a comparative analysis of the suggested solution. As a basis for evaluation and benchmarking tests, a framework for frequent trajectory patterns mining and identification of trajectory outliers in a three dimensional ST trajectory data, extracted from video surveillance cameras, will be implemented. A video from surveillance cameras will be processed in a tracking system, which extracts vehicle trajectories and converts them into vectors containing tracking points. The implemented method needs to be evaluated in terms of accuracy, performance, and an improvement to increase the accuracy of results in context of input data particularities needs to be suggested.

In order to achieve the main objectives, following sub-tasks need to be performed:

- Perform state-of-the-art review of existing approaches and choose a method to implement frequent trajectories extraction and anomalies detection;
- Investigate and suggest an improvement of the chosen algorithm to increase accuracy of results for data from video surveillance cameras;
- Implement a framework with the selected algorithm to test the proposed approach and evaluate obtained results;
- Perform evaluation of implemented algorithm in terms of performance and accuracy.

In this thesis, we will focus on following types of anomalies:

- Anomalous trajectories with anomalous spatial information. This category covers trajectories with abnormal spatial behavior, such as illegal U-turns on the intersection, double solid line crossing, driving in an opposite direction.
- Anomalous trajectories with anomalous spatiotemporal information. This type corresponds to situations where the spatial information can be considered as a normal, but adding a temporal information converts the trajectory into an abnormal one, for example: moving with an anomalously high or low speed, unexpected, emergency stops.

1.2 Thesis Structure

The rest of this thesis work is structured as follows. The whole paper is organized into 7 parts. Chapter 2 introduces the background and terminologies used in the thesis work. Chapter 3 performs the State-of-the-art analysis of existing approaches. Chapter 4 presents the concept

of the suggested approach and an implemented framework. Chapter 5 describes input data structure and input data processing and provides the detailed description of the implementation part. Chapter 6 presents experimental results and discusses evaluation of implemented approach. Chapter 7 gives conclusion and discussions on possible further perspectives.

2 Basic Knowledge

This chapter is intended to give a background information, introduce useful definitions and basic concepts of approaches used in following chapters. Input data sources and related challenges will be discussed.

2.1 Input Data Sources

Tasks of frequent trajectories identification and outliers detection can be applied to different data sources, for example: GPS (Global Positioning System) devices and sensor networks, then trajectory data is collected by sensors on moving objects, which periodically transmit information about location over time, or video traffic surveillance cameras. This work will focus on working with latter type of input data sources.

Video data from enforcement cameras is considered as a raw data and is not used directly as an input for implemented system. Raw video processing is done in a stand-alone tracking system. Tracking system takes raw video from enforcement cameras and handles it to perform the objects detection and converting the trajectory into a number of tracking points on images. Tracking points, containing such information as vehicle ID, timestamp, spatial coordinates, are used as an input.

2.2 Trajectory Definition

Trajectories can be described as multi-dimensional sequences containing a temporally ordered list of locations along with any additional information [7]. So, since a trajectory, denoted as τ or T, represents consecutive positions of a moving target object in temporal domain, in a case of a single-camera surveillance data, it can be defined as:

$$\tau = (x_1, y_1, t_1), (x_2, y_2, t_2), \ldots, (x_n, y_n, t_n) \tag{2.2.1}$$

where (x_i, y_i) denotes the position of the target object in the image at time t_i [5]. According to this, trajectories can be represented as a sequence of 3D points, where 2D object is for geometric coordinates and the third dimension stores the time [11].

Generally, trajectory data is raw and contain only minimum information such as position and time as well as the identifier of the tracking object. This information can be easily augmented by such detailed information as speed, acceleration and direction, since they can be extracted from the initial trajectory data [12].

2.2.1 Trajectory Anomaly Definition

Twenty-four-hour recording video surveillance cameras produce massive amounts of data about moving objects, and that increases the possibility that along with the normally behaving

objects some of the moving objects will demonstrate abnormal behavior. Such exceptional behaviors can also be named as outliers, anomalies, abnormalities, exceptions, novelties or deviants [13][14]. Notwithstanding that no standardized way of deviation characterization exists, in statistics following definition can be found [15]:

> "An outlying observation, or outlier, is one that appears to deviate markedly from other members of the sample in which it occurs".

Trajectory anomalies can be described as traffic flow patterns, which significantly deviate from some normal behavior pattern or, in other words, inconsistent with the rest of traffic behavioral patterns. Anomalous trajectories are supposed to have great local or global difference with the majority of trajectories in terms of a chosen similarity metric [6].

The process of outlier detection is intended to reveal unusual patterns that drastically differ from majority of samples in order to process them further in an appropriate way [13]. Also anomalous to normality activity patterns ratio should be relatively small in order to be able to distinguish abnormalities from the dominating normal patterns.

2.2.2 Trajectory Anomalies Classification

According to the literature, trajectory anomalies can be categorized as follows [14][16][17]:

- *Point anomaly* – represents the simplest type of anomalies. Corresponds to an individual data instance which is considered as an abnormal one with respect to the rest of data, since it deviates significantly from all other samples in a data set. For example, a non-moving car on a busy road.
- *Contextual anomaly* – corresponds to a data instance which is considered as anomalous in a specific context, but as normal otherwise. It can be also understood as a point anomaly in a neighboring of the data point itself. Contextual anomalies are also referred as conditional anomalies and represent the most common group of categories applicable to spatial and time-series data. For example, trajectories can be classified based on the spatial data (coordinates) in the scope of a time. Examples of a contextual anomaly may be trajectories of a vehicle moving with a much higher speed comparing to others in the same traffic flow or of a vehicle driving in the opposite direction.
- *Collective anomalies* – a set of data instances, cooccurence of which as a group is considered as an anomaly with respect to the whole data set, while each data instance individually does not necessarily represent an anomaly. The given definition can be simplified to a set of neighboring point anomalies or context anomalies. Collective anomalies can only be applied to data sets with a relation between data instances.

The other way of trajectory outliers systematization may be dividing them into following categories according to the properties, which were used to perform classification:

- *Spatial trajectory anomaly* – classification process takes into consideration only spatial information of moving object trajectories, such as position coordinates. Examples of spatial anomalies can be illegal U-turns, double line crossing or moving in an opposite direction.
- *Temporal trajectory anomaly* – corresponds to anomalies detected by analyzing only temporal characteristics of trajectories, such as duration, time of moving. For example, a trajectory with significantly long duration or a trajectory appearing at an anomalous time.
- *Spatiotemporal trajectory anomaly* – can be detected by analyzing spatial and temporal information in aggregate. Examples of ST anomalies can be vehicles moving with a considerably high speed comparing with majority of trajectories. Also such anomalies can be detected in the case of a contra-flow traffic systems with a reversing traffic light anomalous trajectories: since for such a line allowed direction changes according to some known or learned schedule, classifier can analyze the trajectory direction together with a temporal information.

In accordance with the second classification, the current work will focus on determining trajectory anomalies of the first and third types (spatial and spatiotemporal trajectory anomalies).

2.3 Challenges

Since ST data differs from other types of data in many aspects, challenges are related to the used data type. A unique quality of it is that ST data instances are not independent and identically distributed, as it is usually assumed to be in many of existing data mining approaches. On the contrary, ST data instances, related to observations made by nearby locations and time, are structurally correlated with each other in context of space and time, and it is important to take into consideration the presence of dependencies among measurements in these dimensions. Consequently, many of the existing data mining approaches are not applicable to ST data, since ignoring the aforementioned characteristics can result in poor accuracy of results. This leads to the necessity of investigating and using different methods for processing such a data to preserve all the relations between information domains [7].

Data uncertainty

It should be noted that the chosen type of an input source leads to difficulties in processing. Since trajectory data is acquired by video enforcement cameras, the first problem is an uncertainty of a location as a result of limitations in measurement accuracy of the used cameras, resolution and the quality of the received images or frame jitter [3]. Moreover, enforcement cameras are placed at some fixed locations on an intersection, because of that one of the particularities of used data are pose and perspective, which can cause challenges while dealing with input video data [16]. The view angle of the camera in respect to the scene ground plane

and distance between the tracked object and camera can affect the performance by decreasing the accuracy of objects detection and tracking: the smaller the angle - the bigger is the problem of determining the center of an object [2][3]. Tracked objects can drive in and out of the camera view, but be still detected while partially visible. This can cause trajectory changes on the borders of the camera scene: displacing and shifting of vehicle trajectories depending on the location of an object in respect to the camera [3]. Quality of the performed trajectory analysis also depends on the input trajectory data, including: quality of used cameras, quality of a tracking system, which converts a video data into a list of trajectories consisting of tracking points.

Moreover, in the current thesis work input data contains trajectories extracted from video cameras without sorting and analyzing, so:

1. input data set can contain both examples of normal and anomalous trajectories;
2. input data does not contain labels.

Aforementioned limitations lead to the necessity to use unsupervised methods to automatically extract normality and abnormality rules from unlabeled data [18].

3 Background and State of the Art

This chapter presents state of the art analysis of existing approaches and discusses advantages and disadvantages of existing approaches. Each section is concluded with a summary, in which the chosen approach is specified and a short argumentation is given.

3.1 Anomalies Detection Techniques

According to [14], the task of outliers detection, which is a main objective task of this work, has been an object of interest and studies of research community originating from 19^{th} Century. Nowadays a huge variety of different techniques for solving the task of detecting outliers and abnormalities in video traffic data are presented, and these approaches can be classified in various ways.

For example, data mining techniques in general and anomaly detection techniques as well as clustering approaches, which are one of the ways to solve the task of outliers identification, particularly can be classified as supervised, semi-supervised and unsupervised on the grounds of the manner of labeling the input data] [14][17]:

1. *Supervised*. Input data used for training contains labels for both normal and anomalous instances. As a result the algorithm can build models for both normal and abnormal classes;
2. *Semi-Supervised [14]* or *Weakly-Supervised [5]*. Input training data set contains class labels only for normal data instances. Such techniques are more widely used than supervised approaches, since anomalous data instances are usually not predictable and random and it is difficult to provide examples to cover all possible anomalous events;
3. *Unsupervised*. Does not require input data to be labeled neither for normal nor anomalous data. Such algorithms are based on the expectation that normal data instances are significantly more frequent than anomalous ones in the test data set and therefore are not applicable when this assumption is disrupted.

Alternatively, according to surveys done by Chandola in [14], Kumaran in [16] and Malik in [17], anomaly detection techniques can be classification based, nearest-neighbor based, clustering based, statistical and etc. The following offers the short overview of mentioned groups.

Classification based

The main concept of these methods lies in using a classifier which firstly learns to distinguish inliers and outliers and then classifies each input instance [19]. Such techniques consist of training and testing phases. Training, or learning, phase supposes learning a classifier model

from a training data set, containing labeled data instances. The learned classifier is then used to classify an input trajectory as normal or anomalous by assigning a class label in a testing phase.

Depending on how testing data instances are labeled, all classification based anomaly detection techniques can be one-class or multi-class. The first type assumes that all training data instances are normal and are labeled as one class. During training phase model learns a discriminative boundary around normal instances, and a trajectory, which is not aligned with the learned normal class description, is considered as an anomalous. Single-class Support Vector Machines (SVMs) is the most commonly used classification based approach, which is applicable to the task of anomalous trajectory detection, as it was proposed by Piciarelli *et al.* [20][21]. However, this approach requires trajectory vectors to be the same length. Since raw trajectory data is usually contains different amount of trajectory points due to different speed of moving objects, it is necessary to preprocess raw trajectories to normalize them to vectors of the same length [21]. Moreover, SVMs become highly time and memory consuming while working with huge amounts of multi-dimensional data [22].

The latter category supposes learning multiple classes during training step and then using a classifier to review the input trajectory for compliance with each learned class. In literature different descriptions of training phase and training data labels are given. According to [14], training data contains only normal data instances with corresponding normal class labels, and during training phase model learns multiple discriminative boundaries around each class of normal instances. A trajectory, which is aligned with none of the learned normal class descriptions, is considered as an anomalous. In other words, an anomalous trajectory will not be accepted by neither of the classifiers. In [16] it is assumed that model is learned using training data containing labels for normal and anomalous classes. Therefore, a classifier can classify an input trajectory as belonging to a normal or anomalous class.

The advantage of two-phased classification based algorithms is a fast testing phase due to precomputed classifier model used to classify each input instance. Also such algorithms can perform well in cases when anomalous data instances form a class or cluster [19]. However, the training step requires accurately labeled training data, which is often not available.

Nearest-neighbor based [14] or Proximity / Distance based [16][19]

Proximity based approaches decide whether a data instance is normal or anomalous based on how close or far is it located with respect to neighbors [16]. Nearest-neighbor and density based approaches are based on the assumption, that «normal data instances have dense neighborhood, while anomalous data instances occur far from their closest neighbors» [14].

In order to be able to compare the surrounding density for an instance under consideration with the density around its local neighbors, a distance (dissimilarity) or similarity measure between two data instances needs to be specified [19]. By virtue of an anomaly score calculation method, techniques can be grouped into two categories: 1) the anomaly score is calculated as a

distance of a data instance to its k^th nearest neighbor and 2) to compute the anomaly score the relative density of each data instance is being computed [14].

These approaches has several disadvantages. First of all, in comparison with classification based anomalies detection techniques, the computational complexity of the testing phase is considerably higher, since nearest neighbors are computed by computing the distance for each test data instance with all instances from either testing and training data. In case of multi-dimensional trajectory data, the task of distance computation becomes even more complicated. Moreover, the accuracy of labeling decreases when the main assumption is violated: when normal instances have sparse neighborhood or anomalous instances have dense [14].

Clustering based

Clustering is an efficient approach aimed to group data instances into different classes, called clusters, based on their similarity in such a way, that objects in one cluster are similar to each other and dissimilar to objects in other clusters [10][22]. ST clustering supposes grouping objects on the ground of their spatial and temporal similarity. To compare data instances before grouping them into clusters, similarity or distance between them needs to be measured.

There are three types of clustering based anomalies detection techniques with following assumptions: 1) normal data instances are associated with a cluster, while anomalous data instances are not associated with any cluster, 2) normal data instances are close to the cluster center, while abnormal instances lie far away from the closest cluster center and 3) normal data instances lie in large and dense clusters, while anomalies are associated with sparse clusters or clusters with a small cardinality [14][16]. Techniques of first type can be implemented using one of the clustering methods which do not require every data instance to belong to some cluster, for example DBSCAN [23]. Algorithms from second group consist of two phases: 1) data clustering and 2) calculating an anomaly score for each data instance. Techniques of the latter type require a threshold for cardinality size and/or density of a cluster to be defined to decide whether a cluster refers to normal or anomalous data.

The necessity to compute distance between trajectories in some of the clustering based approaches makes them similar to neighbor based approaches. As it is stated in [14], techniques are different in the way they process instances: in clustering based techniques each instance is evaluated with respect to the corresponding cluster, while in neighbor based techniques each instance is being inspected with respect to its proximate neighborhood. Consequently, the selection of distance computation method plays an important role and affects results and performance significantly.

On the other side, dividing all training data into groups makes clustering based algorithms similar to classification based algorithms. Though in classification based approaches class is assigned based on given labels, while in clustering based approaches classification is not given in advance [19].

One of the main advantages of clustering based techniques is the ability of majority of them to run in an unsupervised manner. For the case of TVS-based trajectory data acquisition the unsupervised learning methods are the most appropriate, because labeling hours of video data is a highly time-consuming task. Also, manual labeling of input data can lead to errors due to human operator intervention.

Moreover, clustering based techniques are adjustable to work with complex data types because of adaptability of clustering algorithms. However, at the same time they are computationally expensive and are used primarily for relatively low dimensional data, highly dependent on the used clustering algorithm and can not effectively deal with situations when anomalies form significant separate cluster groups [14].

Model based [16][19] or statistical [14]

The main concept of model based algorithms is that they represent the data as a set of parameters to create the model of a normal behavior. As an advantage, model based approaches do not ask the user to provide any input parameters, because all the parameter values can be derived from the data. Statistics based approaches can be considered as a subcategory of model based approaches, they are treated as one of the earliest algorithms and can be used as a basis by the various outlier detection techniques [17]. As it is stated in [14], the main idea of statistical approaches is that data instances occurring in high probability regions of a stochastic model assumed to be normal, while data instances from the low probability regions refer to anomalies. So, statistical approaches are based on using statistical stochastic model to fit to the given data and then applying a statistical inference test, also called discordance test, to decide if a data instance is normal or anomalous. It comes from the main concept that «based on results of applied statistical test, anomalies have low probability to be generated from the learned stochastic model» [14].

Statistical techniques in turn can be parametric or non-parametric [17]. In parametric approaches the normal data is supposed to fit the parametric distribution and probability density function with estimated from the given data parameters [16]. One of the advantages of parametric techniques is that the data size does not affect the model: models grow only depending on a model complexity. However, the necessity to fit the data into some preselected distribution model complexifies and limits the application of such approaches: it is difficult to fit the data to one distribution. In this case it is possible to use a multiple-distribution model to match some clusters of the data with particular distributions [19]. One of the most known examples of parametric methods is Regression Method [17].

By contrast to this, non-parametric approaches are based on using non-parametric statistical models with structures, which are not defined in advance: the given data is used to determine the structure dynamically. Such approaches do not suppose making assumptions about the statistical distribution of the data [17].

Since statistical approaches are based on fitting a statistical model, the choice of it significantly affects results, computational complexity and performance. Nevertheless, the main assumption of statistical approaches that the data comes from a particular distribution can not be always satisfied, specifically for the case of a multi-dimensional data [14].

Summary

Based on the given description of different approaches and their advantages and disadvantages, it was decided to focus on clustering based anomalies detection approaches for several reasons:

1) they can work in an unsupervised mode without a human intervention and do not require the input data to contain labels,
2) input data is allowed to contain anomalous trajectories,
3) clustering method can be easily applied to such a multi-dimensional data as trajectories by defining a suitable similarity measure.

That means that a clustering method and a similarity measure need to be specified.

3.2 Clustering Approaches

Clustering is a highly researched form of data mining, and huge variety of clustering methods has already been proposed in literature [10]. State-of-the-art analysis of related research papers revealed that all traditional clustering approaches are usually categorized into five types: partitioning, hierarchical, density-based, model-based and grid-based methods [5][10]. Next paragraphs will briefly discuss each of the categories with highlighting main assumptions and concepts.

Partitioning, or Partition-based, methods

Such methods are based on partitioning the trajectories data set randomly and then regrouping clusters by reassigning objects from one partition to another to minimize the objective function. They require the predefined parameter, usually denoted as k, which determines the amount of final clusters, or partitions, to be created. The main requirement is that number of partitions must be smaller than number of initial data points, since each partition forms a cluster, that means that it must be non-empty and contain at least one data instance, and each data instance must be included into exactly one cluster.

One of the most well-known examples of partitioning clustering algorithms is a K-Means algorithm, where firstly k cluster centers are initialized randomly and then data points are iteratively reassigned to the closest clustering center based on the discrepancy to minimize the clustering error [24]. The clustering error is defined as the sum of the squared Euclidean distances between each data set point and the corresponding cluster center [25]. The process is stopped when there are no more changes in clustering centers.

The disadvantages of the traditional K-Means clustering method are inability to form clusters of arbitrary form, dependence on initial random cluster centers initialization and high memory consumption [10]. Also finding an appropriate partitioning technique is a challenging task.

Hierarchical methods

In hierarchical based methods the given data set is decomposed into multiple levels to organize a hierarchical tree of clusters. The resulting hierarchical structure can be depicted as a tree [24].

There are two different ways of hierarchical decomposition: 1) the bottom-up (combining) and 2) the top-down (split, divisive) decomposition. They refer to agglomerative and divisive (split) clustering approaches respectively [26].

Agglomerative hierarchical clustering algorithms start by assigning each data instance to a distinct singleton-cluster, so the number of initial clusters is equal to the exact amount of data instances in input data, and then continue uniting clusters based on theirs similarity until all the initial clusters are merged into one single cluster or into predefined amount of clusters [27]. This is done by repeatedly executing following two steps: 1) identifying the two closest clusters and then 2) merging these two clusters [26]. Proximity matrix is used to store similarity measurements between clusters and is being updated on each step by computing distances between the new cluster and the other clusters.

The divisive hierarchical clustering algorithms work in a reverse manner: initially all data instances belong to one cluster and then step by step clusters split into smaller clusters until all of them become singleton clusters or until satisfying some predefined end condition.

Hierarchical clustering is supposed to be simple, but it is necessary to choose between agglomerative and split methods. Divisive clustering is more expensive in computation, therefore, it is less common than agglomerative approaches. Irreversibility of both splitting or uniting processes in traditional hierarchical clustering algorithms is also a particularity of such algorithms [10].

Since approach includes clusters joining, a significant task of agglomerative clustering algorithms is defining and computing the similarity or distance between clusters. This similarity can also be referred to as an inter-cluster or between-cluster distance. For the case of single-trajectory clusters the similarity between them is simplified and is equal to the similarity between respective trajectories. For multiple-trajectory clusters the similarity is computed according to a chosen linkage method. In literature following linkage methods are given as mostly common: single link, complete link, average link [24][28]. The choice of the linkage method depends on the application domain [26]. In the case of the single link distance between two clusters is defined as the minimum distance between two trajectories in these clusters, that means that the similarity between of two clusters is determined by two closest trajectories. The complete link linkage method implies taking the maximum distance between two trajectories in two clusters as an inter-cluster distance, so it is defined using the farthest distance of trajectory pairs. The

average link supposes calculating averaged paired distance between all trajectory pairs in these two clusters.

A convenience of agglomerative hierarchical clustering approaches is that they do not require the number of resulting clusters to be predefined, so they are appropriate for clustering vehicle trajectories, because number of clusters of normal or anomalous trajectories is not known in advance. On the other side, hierarchical clustering methods require the termination condition to be predefined to stop the merge or division process [23]. For example, the critical distance between all resulting clusters can be used as a termination condition in agglomerative hierarchical approaches. However, the most well-known disadvantage of hierarchical clustering algorithms is that they are not robust and can suffer from noise and anomalies.

Density-based methods

In comparison with the partitioning and hierarchical clustering approaches, density-based methods objects inspect similarity based on the density of the data [22]. The area is being added to the nearest cluster, while density of the points in the area remains greater than the predefined threshold [10]. Clusters form dense regions of objects and they are separated by sparse regions with low density.

The main advantage of density-based clustering approaches is that they are able to form clusters of arbitrary forms, extend beyond spherical [10]. Also they are appropriate for clustering huge data sets of trajectories in an unsupervised manner and do not require the amount of clusters to be known in advance [5][22]. However, the results quality highly dependent on the amount of trajectories in training data set, available for analysis.

The most well-known and commonly used density-based algorithm is a DBSCAN, proposed by M. Ester *et al.* in [23]. According to it, input data points are categorized as follows: core data, density-reachable data and outliers based on parameters ε, *minPts* and the density threshold. Neighbor parameter ε (distance of neighborhood) and *minPts* specify the maximum remoteness and minimum amount of satisfying points while choosing the core points: at least *minPts* points must be present within distance ε from the core point, these points are marked as directly reachable from the chosen core point. Aforementioned parameters need to be predefined by the user, but it is difficult to determine them correctly. Each cluster must contain at least one core point. Points are denoted as anomalous if they are not reachable from any of the other points.

Shrinkage-based or Grid-based methods

The main idea of grid-based algorithms lies in applying a multi-resolution grid data structure: the data space is quantized into a finite number of cells (units) that form a multi-resolutional grid structure. Each cell stores summary information about data objects within its subspace [22]. Since clustering operations are performed on the created grid, and also important trajectories characteristics can be computed in each of the spatial grid cells, the quality of data compression

influences the quality of results significantly [7]. Density of closely located dense cells can help to determine clusters. A trajectory can be considered as an anomalous if it differs from the expected trajectory in a number of covered grid cells [22].

The main advantage of grid-based clustering algorithms is an improved performance: increased processing speed and processing time becomes independent on the size of the data set, only the number of cells in each dimension in the quantized space affects the processing time [10].

Model-based methods

In comparison with the above methods, which analyze distance among data objects, in model-based approaches data is supposed to be generated by a mixture of probability distributions, where each component of mixture represents a cluster. So a mathematical model is assigned to each cluster, and then method attempts to find the best fitting data for the chosen model. In this way such methods seek to increase the adaptability between given data and some statistical models [10][22]. The idea of model-based algorithms is that in order to locate clusters they describe the spatial distribution of the input data points by building density functions. The model-based approaches are typically used in feature-specific clustering and depend on the selected features and model [5].

It is emphasized, that model-based approaches show good performance while working with complex data types. This category usually includes statistical and neural network methods [10].

Graph-based methods [7]

Another category of clustering methods in application to vehicle trajectories data. Liu *et al.* in [29] presented a graph-based approach to solve the problem of detection of outliers in traffic data streams. A graph structure was used to store the traffic: nodes represent regions while edge weights depict the traffic flow. Edge anomalies in the graph denote the traffic abnormalities, and causal outlier tree can then be used to further analyze these outliers to find causal interactions.

Another higher-level classification of clustering methods can consist of only two sub-classes on the ground of properties of generated clusters: hierarchical and partitioning approaches [24]. Hierarchical algorithms group objects into clusters from singleton cluster to cluster containing all data instances or in a reverse direction. While partitioning clustering algorithms divide given data set into a predefined number of clusters in a single-layer structure.

In order to perform clustering, the similarity between two trajectories needs to be defined. Different existing distance measures will be reviewed in following paragraphs.

Summary

Based on the given description of clustering approaches, their limitations, advantages and disadvantages, it was decided to focus on a hierarchical clustering approach, more specifically on an agglomerative hierarchical clustering, because it can deal with limitations of a given input

data, that are: absence of input labels, unknown number of resulting clusters, presence of both normal and anomalous trajectories in input data.

3.3 Distance and Similarity Measures

As it was mentioned before, clustering based approaches require a similarity measure to be defined between two trajectories. Apart from that, distance and similarity measures are also used to compare a trajectory with a cluster or a pair of clusters between each other. A similarity measure highly dependent on the format of a trajectory. A trajectory data, represented as a multidimensional data, can contain quantitative or qualitative features, continuous or binary. In such a classification, distance measure functions are more appropriate to work with continuous features, while similarity measures – with qualitative features [24]. Input trajectory-vectors in this work contain spatial information along with temporal, which can be termed as qualitative continuous data. That means that distance measure functions are more appropriate in this case. Moreover, distance and similarity functions can be classified as 1) working with raw representations of trajectories without any preprocessing steps and 2) working with preprocessed trajectories representations. Preprocessing can include unifying the length of trajectories or reducing the dimensionality of trajectory-vectors [28].

Some of the most known and widely used traditional similarity measures are following: Euclidean Distance, Fréchet Distance, DTW, LCSS.

Euclidean Distance

Euclidean distance between two trajectory vectors is calculated as a sum of squared differences of corresponding spatial coordinates [18]:

$$d_{ij} = ||T_i - T_j||_E = \sqrt{\sum_{k=1}^{m}((t_{i_x}^k - t_{i_x}^k)^2 + (t_{i_y}^k - t_{j_y}^k)^2)} \tag{3.3.1}$$

where both trajectories consist of *m* tracking points and are represented by two-dimensional vectors $T_i = \{t_i^1, t_i^2, \ldots, t_i^m\}$ and $T_j = \{t_j^1, t_j^2, \ldots, t_j^m\}$. Tuples $(t_{i_x}^k, t_{i_y}^k)$ represent spatial coordinates for a *k*-th tracking point of *i*-th trajectory from a data set.

However, Euclidean distance works only with trajectories with equal number of tracking points. Since usually vehicles move with different speed and behavior, trajectory length is always different and that means that raw trajectories need to be preprocessed and reduced to the same size [28]. Also, traditional Euclidean distance requires two-dimensional data, meaning that it is not able to process temporal information, and is dependent on the trajectory direction: the reversed direction can cause incorrect distance measurement, that in its turn leads to errors in clustering. Also, it fails while working with trajectories moving in a similar way but with different speeds and in the case of different sampling rates [30].

Fréchet Distance

Fréchet Distance is based on Euclidean distance. It considers the positional and sequential relationship of trajectory points while calculating the similarity. The main idea of this approach is computing Euclidean distance for each pair of points from two trajectories and then designating the maximum Euclidean distance as a Fréchet Distance between them [10][31]. However, since only the maximum among distance is considered, the approach is sensitive to the presence of outliers.

DTW

Dynamic Time Warping (DTW) is one of the algorithms for measuring the similarity between two temporal time series sequences, which may vary in speed. The objective of time series comparison methods is to produce a distance metric between them two. DTW method aims to find an alignment between time-dependent sequences, such as trajectories, and is able to process trajectories of different lengths [10].

According to [10], DTW distance is calculated as follows (Formula 3.3.2):

$$D_D(T_i, T_j) = \begin{cases} 0 & m = n = 0 \\ \infty & m = 0 \text{ or } n = 0 \\ dist(a_i^k, b_j^k) + min \begin{cases} D_D(Rest(T_i), Rest(T_j)) \\ D_D(Rest(T_i), T_j) \\ D_D(T_i, Rest(T_j)) \end{cases} & \text{others} \end{cases} \tag{3.3.2}$$

where $D_D(T_i, T_j)$ refers to DTW distance between two trajectory segments with lengths m and n, $dist(a_i, b_j)$ means the Euclidean Distance between two trajectory points. Function $Rest(T_i)$ takes the remaining part of a trajectory after excluding the point a_i. It can be seen, that in case of zero-length trajectories the DTW distance is equal to 0, for the case then only one of two trajectories is non-empty, the distance between them is considered to be infinite. For two non-empty trajectories, the minimum distance between them is calculated in a recursive way.

Though the important advantage of the DTW method is its ability to process trajectory vectors of distinct lengths, DTW distance is not robust to noise and requires trajectory points to be continuous. Also DTW distance computation is highly time consuming and complex due to necessity to compare distances between each pair of trajectories.

LCSS

Longest Common SubSequence (LCSS) distance tries to match two trajectory sequences based on the longest common sub-sequence between them. The LCSS algorithm works with discrete values and calculates the largest number of equivalent points between the two

trajectories. The task of finding the longest common sub-sequence is usually solved recursively [10]: possible translations, or shiftings, are calculated in each dimension and used to provide the maximum LCSS [32]. The basic idea of an LCSS distance is that it allows two trajectories to stretch. In comparison with DTW and Euclidean distances, LCSS enables some elements to remain unmatched [33] and, in comparison with DTW, LCSS is more robust against presence of outliers [34].

The LCSS distance is calculated according to the Formula 3.3.3 [28]:

$$D_{LCSS}(T_1, T_2) = 1 - \frac{LCSS_{\delta,\epsilon}(T_1, T_2)}{min(m, n)} \tag{3.3.3}$$

where m and n are lengths of trajectories T_1 and T_2 respectively. $LCSS_{\delta,\epsilon}(T_1, T_2)/min(m, n)$ can be also referred to as an LCSS similarity and takes value between 0 and 1.

The $LCSS_{\delta,\epsilon}(T_1, T_2)$, the longest common sub-sequence between trajectories, represents the number of matched trajectory points between trajectories T_1 and T_2 and is defined as follows (Formula 3.3.4):

$$LCSS_{\delta,\epsilon}(T_1, T_2) = \begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ 1 + LCSS_{\delta,\epsilon}(Head(T_1), Head(T_2)) & \begin{array}{l}(\text{if } |t_{1_{x,m}} - t_{2_{x,n}}| < \epsilon \\ \text{and } |t_{1_{y,m}} - t_{2_{y,n}}| < \epsilon \\ \text{and } |m - n| \leq \delta)\end{array} \\ max \begin{cases} LCSS_{\delta,\epsilon}(Head(T_1), T_2) \\ LCSS_{\delta,\epsilon}(T_1, Head(T_2)) \end{cases} & \text{otherwise} \end{cases} \tag{3.3.4}$$

As it can be seen, LCSS calculation depends on two constant parameters: δ (point spacing [32]) and ϵ (point distance [32] or matching threshold [33]):

- parameter δ defines the maximum remoteness in terms of time between two trajectory points in which we can look to match a given point from one trajectory with another. Also can be defined as a value representing the maximum index difference between two input trajectories allowed in calculation [32].
- constant ϵ defines the size of proximity to look for matches in terms of spatial information. According to [32] it is a floating point number which represents the maximum allowed distance between trajectory points in each dimension to consider them as equivalent: difference between X- and Y-coordinates less than ϵ value means that points are relatively close to each other and can be considered as equivalent. LCSS distance is increased by 1 in this case.

Parameters δ and ϵ affect results significantly, therefore, the task of choosing the optimal values for them is challenging and important [28][30]. The $Head(T)$ function is defined to return the first $M-1$ points from the trajectory T, representing the trajectory with the last trajectory point removed. According to implementation given in [32] the LCSS computation, based on a dynamic programming approach, has a complexity of $O((m+k)\delta)$. However, the algorithm requires predefined constant δ and ε parameter values as an input to a method. Also, due to the recursive way of computations, LCSS has a high computational cost [35].

Summary

The LCSS distance is the most appropriate in this work, since it allows the trajectories to contain noise, have different length, objects speed and sampling rates (local time shifts in trajectories) [28]. Moreover, among the aforementioned methods, the LCSS distance is the most robust approach against noises.

3.4 Summary

Related Approach

The aforementioned objective has been investigated and solved in numerous works using different methods. Since in fact normal events are common and dominate the data, and abnormal events are rare and difficult to describe explicitly, many approaches are based on an unsupervised clustering of trajectories. For this thesis work the approach proposed by Ghrab, Fendri, Hammami in [28] was chosen as a basis. It is focused on detection of abnormalities based on a trajectories clustering.

The proposed approach can be described as a two-phase approach with offline clustering to extract frequent trajectories and an online classification of an input trajectory to label it as a normal or anomalous.

The clustering is done in an unsupervised manner using an agglomerative hierarchical clustering algorithm operating on a distance matrix between trajectories. To perform clustering, the LCSS distance is used as a similarity measure. The formulas and description of the LCSS distance are given in the previous section (Formulas 3.3.3 – 3.3.4).

Advantages

One of the advantages of the proposed method is that the chosen similarity measure does not require the trajectories to be of the same length, so that the preprocessing of the trajectories, which is a high complexity process, can be avoided. Moreover, the training data is allowed to contain normal trajectories as well as anomalous: algorithm will extract both normal and anomalous clusters. Dense clusters will represent normal trajectories classes, sparse clusters – anomalous trajectories classes.

Disadvantages

However, the disadvantage of the proposed method is that LCSS distance does not take into consideration such problems of video surveillance as a view perspective and a position of a moving object.

Summary

This thesis work will be intended to investigate an opportunity of increasing the accuracy of results by making epsilon and sigma parameters, which are used to calculate the sigma, adaptable and dependent on the perspective and a distance from the camera. This includes:

- exploring a functional dependency between epsilon and sigma parameters and a distance from the camera,
- evaluating algorithm with different values.

4 Suggested Approach

In this chapter the description of the conceptual model for suggested approach is given. In first sections the conceptual model and the architecture of the solution are discussed. Following sections specify the chosen methods for input data processing and approximation, clustering, outline the main algorithms.

4.1 Framework Conceptual Model

The main objective of current thesis work is analyzing ST trajectory data extracted from videos from surveillance cameras and solving the task of identifying outliers. To solve this task the clustering approach was chosen as a basis: firstly trajectory data is used as a training data to define clusters and model them; clusters are denoted as a normal or abnormal ones based on their density; then classifier marks an input trajectory as a normal or anomalous one.

The contribution of the work is in making an attempt to solve the problem coming from data uncertainty and increase the results accuracy by adapting the algorithm of measuring trajectories similarity: take into consideration the position of moving objects in respect to the camera. For this purpose a framework covering mentioned tasks was implemented with an ability to extract frequent trajectories and then detect anomalous trajectories.

The basic workflow of the framework consist of processing the input data, performing trajectories approximation using a polynomial regression, calculating the similarity matrix between trajectories, then clustering the trajectories and modeling the extracted clusters, identifying normal and anomalous clusters based on their density, visualization of modeled clusters, and finally taking an input trajectory and classifying it as a normal or abnormal one according to the built clusters' models (Figure 4.1.1).

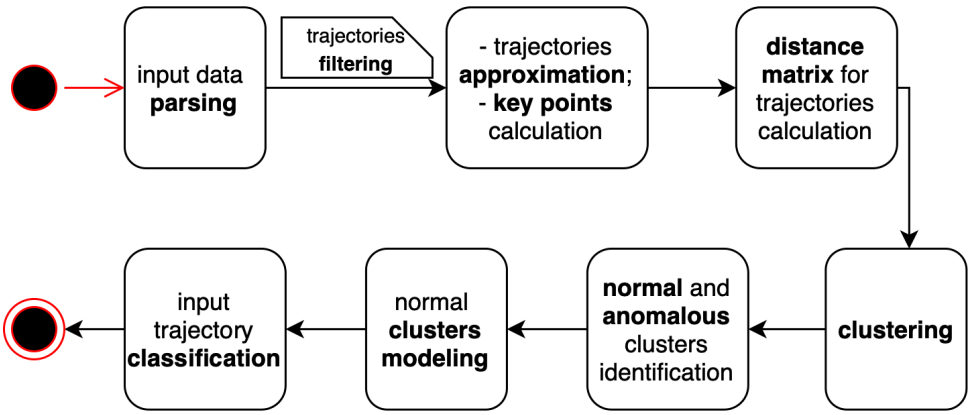


Figure 4.1.1: Basic workflow of the framework

4.2 Framework Architecture

The architecture of the framework is based on an already discussed related work [28] and consists of two phases (4.2.1):

- *offline* to perform clustering and extract frequent trajectories, and
- *online* to classify the new trajectory as a normal or abnormal one.

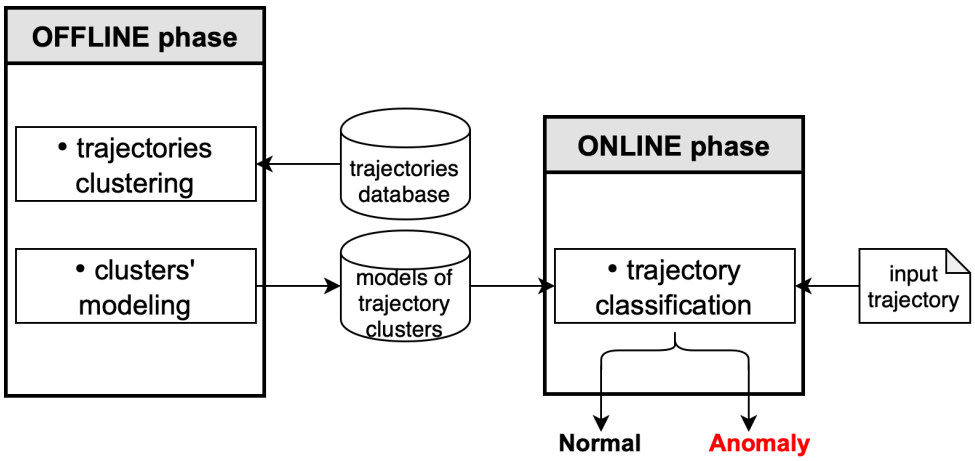


Figure 4.2.1: Two-phased proposed approach

The implemented framework consists of several modules which are responsible for performing particular steps of the aforementioned workflow (Figure 4.2.2):

- *entity* – contains entity-classes for $Trajectory$, $TrajectoryPoint$, $Cluster$ and etc. objects;
- *parsing* – reading a 'txt'-file with input trajectories and parsing them to create $TrajectoryPoint$ and $Trajectory$ objects;
- *csv* – contains logic for reading and writing from/into 'csv'-files, is used to save calculated LCSS measures and load to proceed with clustering;
- *approximation* – performs an approximation of trajectories using a Polynomial regression;
- *visualization* – is responsible for visualization and saving the results, contains methods to read, edit, save $BufferedImage$'s;
- *clustering* – consists of a $Clustering$ class which contains methods to compute LCSS metric values, perform clustering of $Trajectory$ objects and create $Cluster$ objects;
- *exception* – contains exceptional classes hypothetically thrown in the framework (e.g. $TrajectoryParserException$);
- *misc* – contains utility classes needed to store constant values and basic methods.

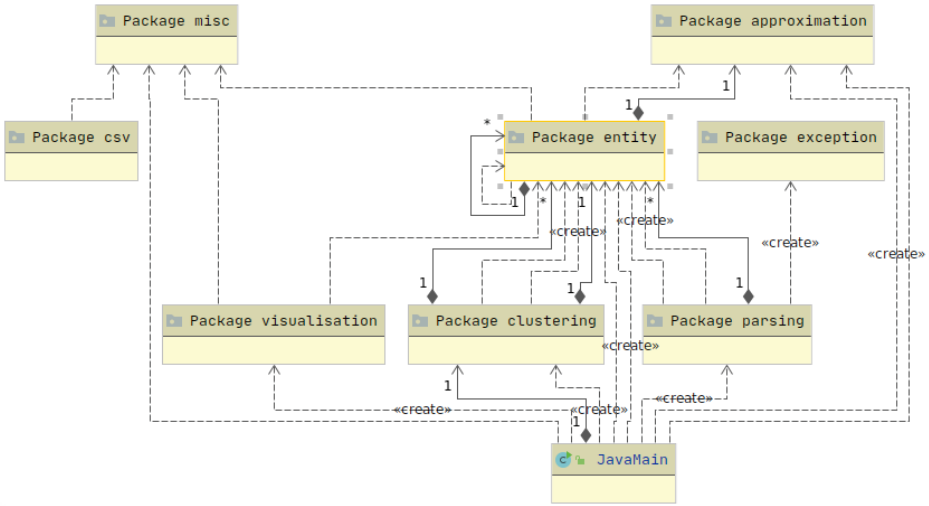


Figure 4.2.2: Architecture of an implemented framework

4.3 Trajectories Analysis

It was mentioned before, that LCSS distance is used as a distance measure between trajectories to perform clustering. LCSS distance implies computing the Longest Common SubSequence between two input trajectories using two parameter values: δ and ε.

4.3.1 LCSS Distance to Measure Trajectories Similarity

Traditionally δ and ε parameters are constant and defined in advance. However, in the developed framework in order to handle uncertainty of trajectory data coming from different position in respect to camera adaptive values of parameters are implemented. Parameters are functionally dependent on position of a moving object on a scene in respect to the camera.

While considering the visualization of trajectories on sample images taken from the cameras, following can be deduced: since the bottom part of the image represent the region located closer to surveillance camera, moving objects on the upper part of the image are more distant from the camera and as a result are more densely located in respect to representation of each other on the image. ε is responsible for the threshold controlling spatial remoteness of trajectory points while computing similarity distance. Consequently, it must be adapted to the remoteness and decrease as a trajectory point gets farther from the camera.

4.3.2 LCSS Parameters Adaptability

The traditional LCSS measure supposes using a constant δ and ε parameters and does not imply making them adaptive. However, one of the objectives in this work is investigating an opportunity to increase results accuracy by exploring a functional dependency between these parameters and a distance from the camera.

Algorithm 1: Description of LCSS distance calculation

Input: First trajectory: T_1,
Second trajectory: T_2,
Temporal remoteness threshold: δ,
Spatial remoteness threshold (default): ε_x, ε_y
Output: LCSS distance for two trajectories
begin
// Initialization
- calculate length of T_1;
- calculate length of T_2;
// LCSS similarity calculation
- take last point of T_1 (TP_1) and T_2 (TP_2);
- get ε_x and ε_y for pair (TP_1, TP_2);
if T_1 *or* T_2 *is empty* **then**
return 0;
else
if *difference between X-coordinates* < ε_x
AND difference between Y-coordinates < ε_y
AND difference between trajectory lengths < δ **then**
- increase LCSS by 1;
- call recursive for trajectories excluding last points;
else
- calculate LCSS for first trajectory and second trajectory excluding last point;
- calculate LCSS for first trajectory excluding last point and second trajectory;
- take maximum between these LCSS values;
end
end
// LCSS distance calculation
LCSS distance = 1 - LCSS similarity / minimum(input lengths)
end

Since camera is placed at a fixed location on an intersection, the problems caused by a perspective can take place. It can be seen from the Figure 5.2.2 depicting the allocation of input trajectories that with distance from the camera, which is placed at the lower fore part of the image, trajectories become more densely located relative to each other (the input data will be described and depicted in the following chapter). In the case of a constant ε value, which controls the threshold while testing the spatial equality of trajectory points, trajectory points of trajectories located far from camera (meaning they appear at the upper part of the sample images and are more densely located to each other) will be incorrectly considered as similar. This can contort the following analysis and skew the further clustering results.

Therefore, the value of ε must change in accordance with the distance from the camera location: ε decreases as a trajectory point gets farther from the camera and increases as a

trajectory point gets closer. Consequently, it can be assumed that the ε and distance are in a negative relation, meaning that in the resulting formula the distance from the camera must appear in a denominator with a some coefficient.

The approach to make the parameters adaptive considered in this work is described in Algorithm 2 and depicted in Figure 4.3.1.

Algorithm 2: Adaptive LCSS parameters calculation

Input: First trajectory point: TP_1,
Second trajectory point: TP_2
Output: Adaptive ε value for TP_1 and TP_2
begin
// Initialization
- compute location of a camera point (CP);
- compute Euclidean distance for two pairs $d_1(TP_1, CP)$ and $d_2(TP_2, CP)$;
- compute corresponding ε_1 and ε_2 values;
- take the max$(\varepsilon_1,\ \varepsilon_2)$ as a final ε to compare TP_1 and TP_2.
end

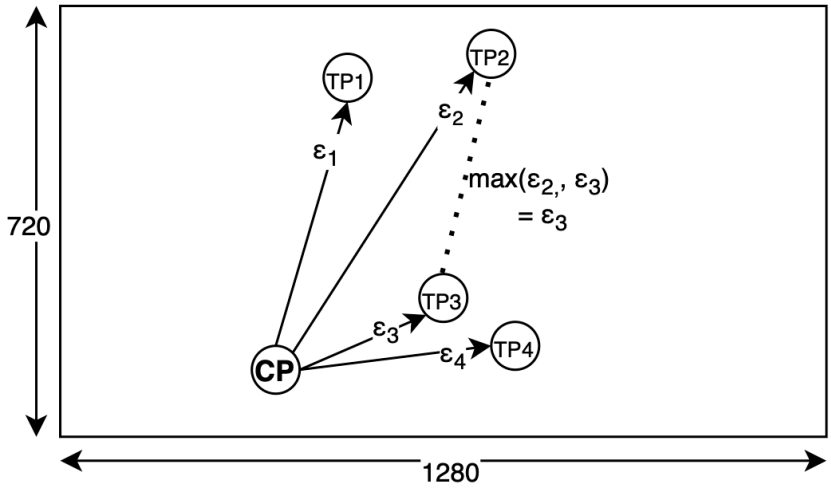


Figure 4.3.1: The principle of adaptive ε value selection

In order to optimize the calculation of a distance to camera point and avoid recalculation of it multiple times during the LCSS computation algorithm, this distance is being calculated in advance and stored for each trajectory point.

4.4 Trajectories Approximation

However, notwithstanding that the LCSS similarity distance works with trajectories of arbitrary lengths and does not natively require the preprocessing of trajectories, the calculation of LCSS measure becomes extremely computationally expensive and time consuming with the growth of the trajectory length because of the recursiveness. Moreover, most of the input

trajectories contain far more trajectory points than needed for further analysis, these redundant information reduces the speed of processing [43]. For that reason it was decided to decrease the size of trajectories in the current work by approximation of trajectory data. This task is called trajectory simplification (TS). The goal of a TS task is to convert a given trajectory, represented by a set of trajectory points $T = (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, into a subset of points $T' = (x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2}), \ldots, (x_{i_m}, y_{i_m})$, where $1 = i_1 < \ldots < i_m = n$ [43].

That leads to the lose of accuracy but allows to get acceptable results in adequate amount of time.

Polygonal approximation techniques can be classified into following classes [43]:

- *min-ε problem* – given an input trajectory T with a length N and integer parameter M, perform an approximation with at most M resulting trajectory points with minimum approximation error ε.
- *min-# problem* – given an input trajectory T with a length N and error tolerance ε, perform an approximation with the minimum M resulting trajectory points satisfying the error tolerance ε.

4.4.1 Curve Fitting

The curve fitting concept is one of the standard approaches to perform approximation [44]. The main task is finding an appropriate relation or law possibly existing between the input (independent) and output (dependent) variables from a given input data set of observed values. And the curve fitting is the process of expressing a relationship between variables in terms of algebraic equations. The main goal of the curve fitting is to find parameters for a model (equation or function) to fit to the experimental data.

Regression Analysis

One of the widely used approaches appearing from curve fitting is a Regression Analysis, which is also considered as a form of predictive modeling approach and, according to the traditional definition, studies the relationship between a dependent variable (response) Y and one or more independent variables X's and tends to find trends in data. In other words it supposes "using the relationship between variables to fit the best fit line or regression equation that can be used to make predictions" [45].

In order to simplify the relationship fitting procedure, it is usually assumed that the independent X variables are measured without an error while the dependent Y variables values are measured with some random error. For the data with a small ratio of the measurement error in an independent variable to the range of values of that variable, it is possible to use the least squares regression analysis with legitimacy [44].

A regression can be linear or polynomial (nonlinear, curvilinear) depending on the function the data is approximated with: linear regression refers to a relationship approximated by a

straight line whereas curvilinear regression refers to a relationship following a curve. Due to a broader range of functions the polynomial regression can work with, it provides better approximation of the input relationship in comparison to linear regression [45]. Even if it is impossible to guess the type of function to use for approximation in advance, plotting the data and analyzing it to find some behavioral pattern, such as linear, quadratic or higher-order dependency, can be useful [44].

Polynomial Regression

The visualization of input trajectory data is given in Figure 5.2.2. As it can be seen from the picture, neither the linear or 2^{nd} order functions can not fit the data properly due to complexity of trajectory forms. For that reason it was decided to focus on approximation using higher-order polynomial regression. The evaluation of polynomial regression with different degrees will be given further and the following discussion and implementation will be intended to find a suitable n^{th} order polynomial equation and parameter values to represent each input trajectory as a 'trajectory function'. Since trajectory data is represented by two-dimensional spatial data along with temporal data and it is necessary to approximate spatial information, x- and y-coordinates will be considered as dependent variables and $time$ will be used as an independent variable. Consequently, polynomial regression will be performed twice with two output polynomial functions representing $x(t)$ and $y(t)$ for each of the input trajectories T:

$$\forall\, T = [\ldots(x_i, y_i, t_i)\ldots] => T(t) = \begin{cases} x = x(t) \\ y = y(t) \end{cases} \tag{4.4.1}$$

Trajectories will be converted from a shape of a list of trajectory points into equations (time functions) defined in a geometrical space, which can represent approximately all of them. Taking key points of the representative polynomial can decrease the size of the trajectory therethrough reducing the total operational cost and computational complexity of LCSS calculation. Moreover, mathematical equations are able to store information in a dense form and apart from other advantages such a data reduction leads to consuming less amount of space and increasing the storage efficiency [44]. Also so called built 'trajectory functions's can provide interpolation and discover the missing data points.

R^2 for Polynomial Regression Evaluation

In order to achieve better approximation the evaluation of polynomial regression results was performed using the Coefficient of Determination denoted by R^2 (also known as a $R\text{-}squared$ score, *Pearson's coefficient of regression*) [36]. R^2 measures the proportion of the response dependent variable variance that can be explained by the regression model with given parameters and is predictable from the independent variable and can be calculated as follows:

$$R^2 = 1 - \frac{SSE}{TSS} = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \overline{y})^2} \tag{4.4.2}$$

where SSE (Sum of Squares due to Error) is calculated as a sum of squared differences between actual y_i, and predicted $\hat{y}_i$ dependent variable values and TSS (Total Sum of Squares) is calculated as a sum of squared deviations of an actual value y_i from a mean $\overline{y}$.

R^2 takes a value between [0, 1], where a value close to 1 indicates that there is a strong relationship between the selected parameters and the model (polynomial equation in this case) predicts the data perfectly [50]. Models with a coefficient value above 0.8 are considered to be adequate; a value of 1 indicates the presence of a functional relationship between the variables.

4.4.2 Ramer-Douglas-Peucker Algorithm

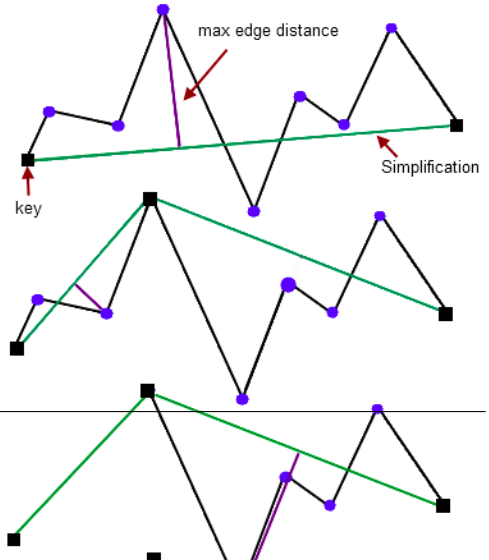
Another approach for solving the TS task by reducing the amount of points in a trajectory curve is Ramer-Douglas-Peucker (RDP) algorithm, also known as an iterative end-point fit algorithm or split-and-merge algorithm.

The main concept of the algorithm is that, given an input trajectory curve represented by line segments (polyline), it seeks to obtain a representative curve with smaller amount of trajectory points (Algorithm 3) [46]. The resulting simplified trajectory consist of a subset of the original trajectory points, with indispensably kept first and last points. The algorithm uses the notion of 'dissimilar', which is calculated as the maximum distance between the original trajectory curve and its approximation.

Figure 4.4.1 depicts the process of a curve approximation using RDP algorithm. The algorithm requires an initial trajectory curve consisting of an ordered set of trajectory points and predefined point-to-edge distance tolerance $\varepsilon > 0$, controlling the remoteness. The input trajectory curve is being recursively divided into segments, while the first line segment (edge) is defined by first and last points as ends. Then algorithm determines the farthermost point (p') for the current line segment ($line$) and processes it in the following way based on the input ε:

- $dist(p', line) < \varepsilon$ – remove all the points, which were not yet marked to be kept;
- $dist(p', line) > \varepsilon$ – mark p' to be kept and call the algorithm recursively on two line segments: 1) from first point to p', 2) from p' to last.

The recursive process continues till all points from the original curve satisfy the point-to-edge tolerance. The simplified trajectory curve can be obtained by choosing only points marked as kept. However, according to the traditional RDP algorithm while using the ε as a point-to-edge tolerance, the limitation for maximum amount of points is not specified and the



resulting simplified curve may contain any number of trajectory points [47]. Such an implementation can be inappropriate in this case, since the main goal of the approximation step is reducing the length of trajectory up to 9 points. For that reason the variation of RDP algorithm was introduced with the use of a point count tolerance instead of a point-to-edge tolerance. It specifies the maximum amount of vertices in the simplified curve. All points of the current approximated trajectory are considered as possible keys and are being processed simultaneously: among them algorithm chooses the point with the maximum point-to-edge distance to be kept.

Algorithm 3: Description of Ramer-Douglas-Peucker Algorithm

```
Input: List of trajectory points: TrajectoryPoint[] trajectory,
       Maximum distance for a dimension: ε
Output: Sublist of trajectory points: TrajectoryPoint[] simplified
RDP ( trajectory, ε ) {
    // Find the point with the maximum distance
    dmax = 0;
    index = 0;
    end = trajectory.length();
    for ( i = 2; i < end − 1; i + + ) {
        d = perpendicularDistance(trajectory[i], Line(trajectory[1], trajectory[end]));
        if d > dmax then
            index = i;
            dmax = d;
        else
    }
    // Initialize empty simplified trajectory points list
    TrajectoryPoint[] simplified = empty;
    // If max distance is greater than ε, recursively simplify
    if dmax > ε then
        // Split and get two results
        TrajectoryPoint[] part1 = RDP(trajectory[1...index], ε);
        TrajectoryPoint[] part2 = RDP(trajectory[index...end], ε);
        // Build the result list
        simplified = {part1[1...part1.length() - 1], part2[1...part2.length()]}
    else
        simplified = {trajectory[1], trajectory[end]}
    end
}
```

Disadvantages

However, traditional RDP algorithm does not provide guaranteed amount of resulting approximation points: the amount of them can significantly vary depending on the input trajectory complexity, number of initial points and defined value of ε.

Douglas-Peucker N Algorithm

To deal with the main disadvantage of the traditional RDP algorithm, *Douglas-Peucker N* algorithm was proposed as its variation. It has 2 main differences [47]:

- using a point count tolerance instead of a point-to-edge distance tolerance as an objective for approximation and
- processing all points of a current simplified trajectory at the same time as possible keys, in comparison with the traditional RDP algorithm, which is based on choosing one random point at each step. The point with the highest point-to-edge distance is considered to be the new key.

Taking into consideration all possible points at any time leads to higher quality of approximation results [47].

The description of the Douglas-Peucker N approach is given in Algorithm 4.

Algorithm 4: Description of Douglas-Peucker N Algorithm

```
Input: List of trajectory points: TrajectoryPoint[] trajectory,
       Maximum amount of points in simplified trajectory: count
Output: Sublist of trajectory points: TrajectoryPoint[] simplified
RDP ( trajectory, count ) {
    - initialize the simplified trajectory with first and last points from original trajectory
    while number of points in simplified trajectory is less than count do
        // for each point from original trajectory:
        for ( TrajectoryPoint: trajectory.points ) {
            - find the corresponding simplified line segment;
            - calculate distance to it;
        }
        - find the point with the maximum point-to-edge distance;
        - add the chosen point to simplified trajectory;
    end
}
```

RDP Algorithm Evaluation using Positional Errors

In order to compare results, the effectiveness and accuracy of RDP algorithm with the results of the Polynomial Regression, the location difference between the original curve and the simplified line will be analyzed. This method is called positional errors calculation. Positional error is being computed for each input original point as the perpendicular difference between

the original point and the respective simplified line segment (Figure 4.4.2) [47]. Lower the positional errors – better are simplification results.

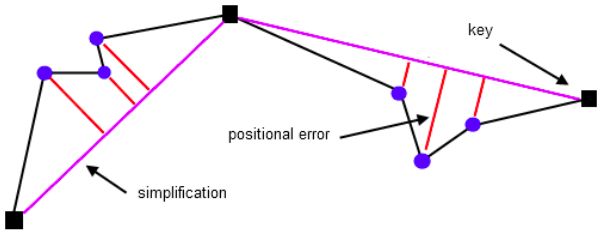


Figure 4.4.2: Positional Errors for RDP evaluation [47]

4.5 Clustering

4.5.1 Agglomerative Hierarchical Clustering

Clustering is done using an unsupervised agglomerative hierarchical clustering approach. The description of this approach is given in Algorithm 5 [28].

Algorithm 5: Description of Agglomerative Hierarchical Clustering

Input: A Database of Trajectories: trajectories
Output: Clusters of Trajectories: clusters
Initialization:
- initialize the clusters with one trajectory in each cluster
Clusters merging:
while *number of clusters is greater than 1* **do**
- calculate similarity matrix D between pairs of clusters based on single linkage approach using LCSS similarity measure;
- find the smallest distance between clusters in D;
- merge two clusters with the corresponding smallest distance into a single cluster;
- remove two merged clusters;

end

As it was already mentioned, agglomerative hierarchical clustering methods suppose clusters joining, which requires the inter-cluster distance measure to be defined. In [28] authors have performed evaluation of different linkage methods, including single link, complete link and average link. Single link, average link and complete link linkage methods consider a minimum, average and maximum distance between two trajectories as an inter-cluster distance respectively and can be summed up as [28]:

$$D_{min}(C_i, C_j) = \min_{T_1 \in C_i, T_2 \in C_j} D_{LCSS}(T_1, T_2), \tag{4.5.1}$$

$$D_{avg}(C_i, C_j) = \quad avg \quad D_{LCSS}(T_1, T_2), \tag{4.5.2}$$

$$D_{max}(C_i, C_j) = \max_{T_1 \in C_i, T_2 \in C_j} D_{LCSS}(T_1, T_2), \tag{4.5.3}$$

where (C_i, C_j) denotes two clusters and (T_1, T_2) corresponds to two trajectories from two clusters respectively.

4.5.2 DBSCAN

The difficulty of DBSCAN algorithm is that it requires the input vectors to be of equal length and was created to cluster the point data. However, it was decided to check its applicability to trajectory data, consisting of two-dimensional *TPs*. To satisfy that requirements, following actions will be performed upon input trajectory data:

- Approximate trajectories using Douglas-Peucker N, since the majority of approximated trajectories have the desired length;
- Flatten two-dimensional trajectory vector into one-dimensional vector with doubled length (Formula 4.5.4);
- Use vector representation of a trajectory as an input data for DBSCAN clustering.

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) => x_1, y_1, x_2, y_2, \ldots, x_n, y_n \tag{4.5.4}$$

Moreover, DBSCAN algorithm requires the ε and $minPts$ parameters to be specified, defining maximum radius of the neighborhood to be considered and minimum number of points needed for a cluster respectively. The evaluation results using different parameter values will be given in Chapter 6.

4.5.3 Normal and Anomalous Clusters Classification

Hereinafter the concept of the cluster cardinality, denoted as $|C|$, is used. The cardinality of a cluster refers to a number of trajectories in that cluster.

Since the main objective of the current work is detection of anomalies by the way of trajectories clustering, following clusters' modeling and finally an input trajectory classification, the concepts of a normal and anomalous cluster must be introduced. Normal cluster is a cluster containing a relatively large amount of trajectories in comparison with others, meaning that such a vehicular behavior, defined by the cluster, is normal and allowed on the current intersection. Consequently, the clusters classification technique needs to be defined.

To perform clusters classification, the analysis of a selection, obtained by calculating the cardinalities of resulting clusters, will be conducted using the order statistical measure, specifically quantile. Quantiles are the way of dividing the whole initial ordered data set into a predefined amount of segments [36]. The α-quantile is a numerical value, which represents the characteristic of the distribution law of a random variable, according to which all values from

this distribution with a fixed probability α (or α^{th} part of all values) do not exceed the specified α-quantile value. In statistical analysis, the most frequently used quantiles are: median (dividing the entire set into 2 segments at the 50th percentile), quartile (dividing into 4 segments by 25, 50, 75%), quintile (5 segments), decile (10 segments), percentile (100 segments). In the current work, it was decided to use the 0.25-quantile (lower, first quartile) as the threshold value.

The general algorithm of normal and abnormal clusters classification in a simplified form is presented on Figure 4.5.1.

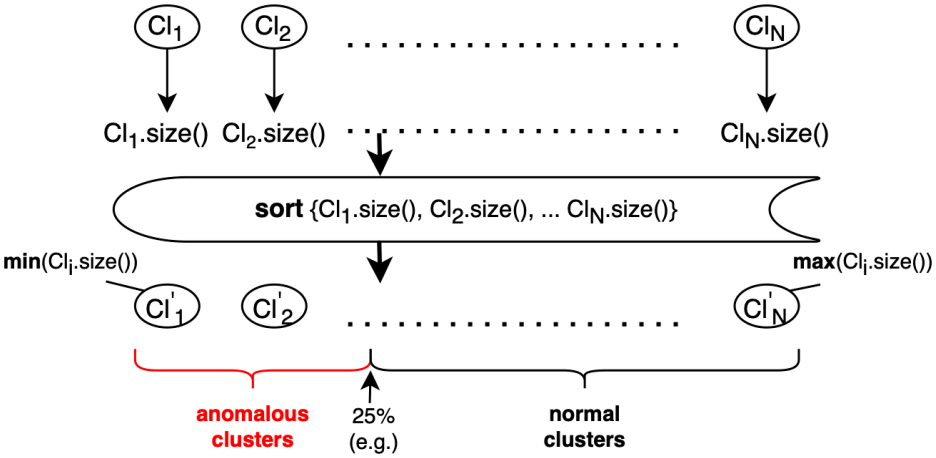


Figure 4.5.1: Normal and anomalous clusters classification

4.5.4 Measuring the Clusters Validity

Since the goal of the current work is finding an optimal adaptive parameter values for similarity measure computation, it is necessary to analyze and compare the results after performing clustering.

According to [38] cluster validity measures can be classified as follows:

- **Internal cluster validation** – the result of performed clustering is being evaluated based on the input data clustered. It is based on an internal information and does not include references to external information.
- **External cluster validation** – evaluation of clustering results is performed in accordance with externally known results, e.g. given class labels. Such validation is not appropriate for unsupervised clustering then no input labels are provided.
- **Relative cluster validation** – evaluation of the clustering results is done by running the same algorithm using different input parameters, such as number of clusters, etc..

At the same time clustering is primarily an unsupervised data mining technique and the input data does not contain data labels. That leads to the necessity to test the resulting clusters in an unsupervised manner.

One of the most widely used and known measures for evaluating clustering algorithms is a Dunn's Validity Index (DI), which was introduced by J. C. Dunn in 1974 in [39]. It is an internal evaluation metric which is intended to identify compact clusters with a small variance between cluster members which are well-separated between each other, meaning clusters are sufficiently distant from surrounding clusters in comparison with inter-cluster variance [40]. Dunn's index is calculated as the ratio between the minimum inter-cluster distance d_{min} to the maximum intra-cluster diameter d_{max} and for k number of clusters can be defined as follows (Formula 4.5.5) [41]:

$$DI = \frac{d_{min}}{d_{max}} = \frac{\min\limits_{\substack{1 \le i \le k \\ i+1 \le j \le k}} dist(c_i, c_j)}{\max\limits_{1 \le l \le k} diam(c_l)} \tag{4.5.5}$$

where minimum inter-cluster distance d_{min} in accordance with the single linkage method refers to the minimal distance between two trajectories from different clusters. Maximum intra-cluster diameter d_{max}, or the largest within-cluster distance in other words, supposes computing the diameter of a cluster as a distance between its two farthermost trajectories [42].

The sample description of a DI index for 3 clusters is given in Figure 4.5.2. According to this sample, Formula 4.5.6 can be written out as follows:

$$DI = \frac{d_{min}}{d_{max}} = \frac{\min(dist_{min}{}^1, dist_{min}{}^2, dist_{min}{}^3)}{\max(diam_{max}{}^1, diam_{max}{}^2, diam_{max}{}^3)} \tag{4.5.6}$$

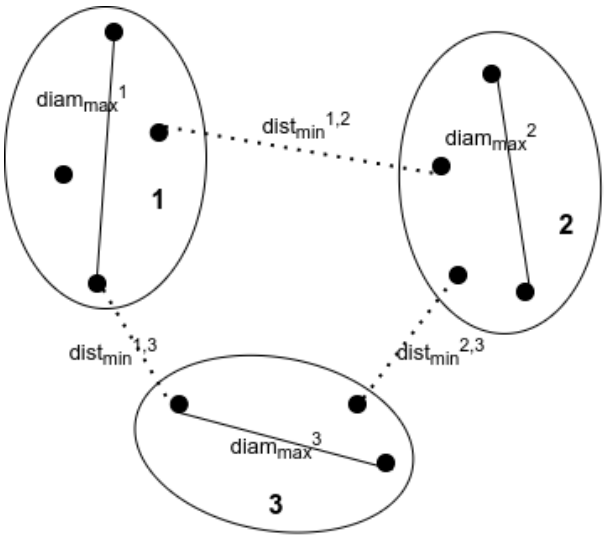


Figure 4.5.2: Explanation of DI

Higher values of the DI indicates the better results of clustering. Trajectories, located far from each other inside one cluster, must be distinguishable from trajectories relating to other clusters. Values of DI close to 1 mean that minimum distances to trajectories from different clusters remain bigger than distance to farthermost trajectories from the same cluster. However,

the computational cost of the DI is highly dependent on the data: the computation cost increases with the increase of number of clusters and dimensionality of the data [38].

4.5.5 Clusters' Modeling

In order to perform a further classification of an input trajectory in a more efficient way, in the related work it was proposed to create cluster models (CM), or path models, for normal clusters. Model of a cluster can be described as a compact representation of it. Models of normal clusters can be considered as patterns of frequent trajectories since they represent the whole cluster of trajectories with the highest accuracy.

Two main concepts of extracting a model from a cluster exist [37]:

- taking a representative trajectory from the cluster. Such a trajectory is considered to be the center of a cluster. An easiest way to specify the representative trajectory is by defining only its centroid (cluster centroid, centroid path). As an extension a centroid can be augmented by a path envelope;
- estimating a model from the trajectories associated with the cluster by the use of probabilistic models, such as the Gaussian observation emission HMMs. Such method requires the trajectories to be preprocessed and is better to apply on probabilistically modeled trajectories.

In comparison with the second case, taking a representative trajectory for each cluster as a model is more simple, moreover, it does not require the trajectories to have the same number of trajectory points [28].

Authors in [28] propose an easy to calculate way of modeling a cluster without any preprocessing of trajectories: a CM is a trajectory least of all distant from others and in view of this can be considered as a cluster center and representation. That means choosing a trajectory with a minimum average LCSS distance to other trajectories in this cluster (Formula 4.5.7):

$$CM(C) = \min_{T \in C} \frac{1}{|C|} \sum_{T' \in C} D_{LCSS}(T, T') \tag{4.5.7}$$

Figure 4.5.3 depicts the main concept of choosing the CM: for each group of trajectories taking as a CM the most qualificative, characteristical trajectory.

Therefore, clusters' modeling is implemented in order to simplify and accelerate the classification step, which, as a result of using the proposed approach of choosing a representative trajectory for each cluster, reduces to calculating the distance between the input trajectory and each of the CMs.

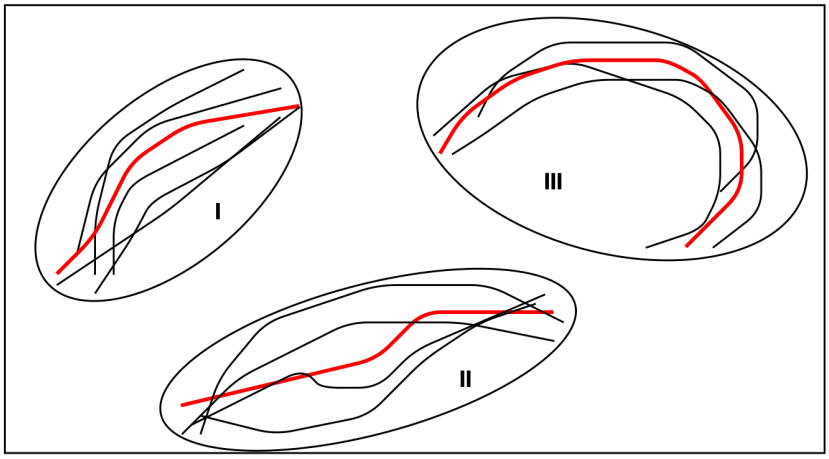


Figure 4.5.3: CM detection

4.6 Trajectories Classification

The last step of the proposed approach is the input trajectories classification and consequently anomalies detection. Since the clusters' modeling is used, an input trajectory classification is simplified to the process consisting of following steps:

1. Approximate an input trajectory;
2. Calculate LCSS distance to each cluster (to its *CM*);
3. Select the closest cluster (normal or anomalous). If the input trajectory does not comply with any of the known clusters, it will be considered as an anomalous.

4.6.1 Anomalies Detection

Consequently, the anomalies detection reduces itself to necessity to classify the trajectories. Anomalies can be detected as:

- trajectories related to anomalous clusters;
- trajectories which are not associated with any of the known clusters.

5 Framework Implementation

In this chapter the description of the framework development is given. It is intended to provide implementation details of a presented concept for a chosen stack of technologies. Based on a workflow of the framework outlined above, separate modules were implemented. Detailed description of each of them is presented in following sections. First sections circumstantiate processing and approximating the input trajectory data. Further sections present the details of the solution implementation to solve tasks of clustering, clusters modeling and input trajectory classification. Since no appropriate ready frameworks providing chosen algorithms were found, all the algorithms implementations, except polynomial regression and polynomial equations solving, were written from scratch and are presented in the Appendix chapter.

5.1 Stack of Technologies

For implementation part of the work Java programming language along with libraries and Apache Maven as a build automation tool were used with following versions:

- Java - 11 OpenJDK
- Apache Maven - 3.6.3
- Commons Math: The Apache Commons Mathematics Library - 3.4.1
- Java AWT, Javax Swing

Java was used as a main programming language of an implemented framework. Commons Math library was chosen for approximation step since it provides implementations for $PolynomialFunction$ and $PolynomialSolver$ classes.

Java AWT (Abstract Window Toolkit) is an API (Application Programming Interface) toolkit for providing a GUI (Graphical User Interface) to a Java application and comes as a part of a Java JDK. In this work Java AWT was used primarily to create objects of $BufferedImage$ with an input image and draw trajectory points on them.

Java Swing as well as Java AWT is a GUI widget toolkit pursuing the same objective of providing a GUI to Java applications and creating window-based applications. However, Swing is more recent and advanced and supports more elaborate set of GUI components than Java AWT. In this work Swing was used to create windows with output images to visualize input trajectories, approximation and clustering results.

5.2 Input Data Processing

This section gives an overview of the used input data format and describes the steps of input data preprocessing.

5.2.1 Input Data Description (Nature of Data)

According to the research done by the US Department of Transportation based on data of Fatality Analysis Reporting System (FARS) and National Automotive Sampling System, nearly 40 percents of all the reported in 2008 year crashes were road intersection related [48]. Consequently, cross-road transport activity analysis is significantly important nowadays in context of safety, and identifying unsafe vehicular trajectories, which violate traffic rules, may be one of the steps towards improving the statistics.

In the presented work video from enforcement cameras is used for training and testing. Test videos are captured using the Intellectual Transportation Systems implemented on four different Kazan crossroads:

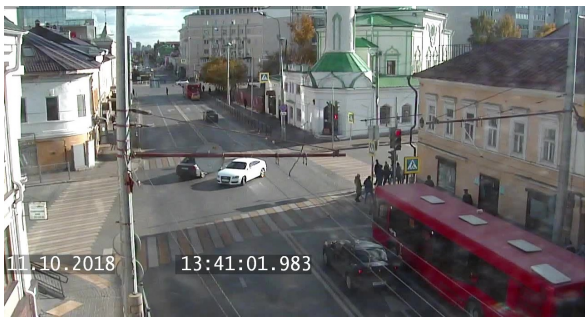
1. An intersection of Pravo-Bulachnaya and Puschkina streets, $1.txt$ (Figure 5.2.1a).
2. An intersection of Nesmelova and Kirovskaya Damba streets, $2.txt$ (Figure 5.2.1b).
3. An intersection of Moskovskaya and Galiaskara Kamala streets, $3.txt$ (Figure 5.2.1c).
4. An intersection of Moskovskaya and Parizhskoy Kommunyi streets, $4.txt$ (Figure 5.2.1d).

Each crossroad corresponds to a 4-way intersection and is equipped with a single monitoring camera. Sample pictures from surveillance cameras are given below on Figure 5.2.1.

(a) $1.txt$

(b) $3.txt$

(c) $4.txt$



(d) $4.txt$

Figure 5.2.1: Input data sources, intersections 1-4

Input data files contain 624, 211, 231, 237 vehicular trajectories for the each of the aforementioned intersections respectively.

By a trajectory anomaly we understand vehicle trajectories through the crossroad, which remarkably differ from majority of common, known trajectories. For example, if no turning to the right from the left line is allowed, such a behavior will be unknown and such a trajectory must be considered as an anomaly.

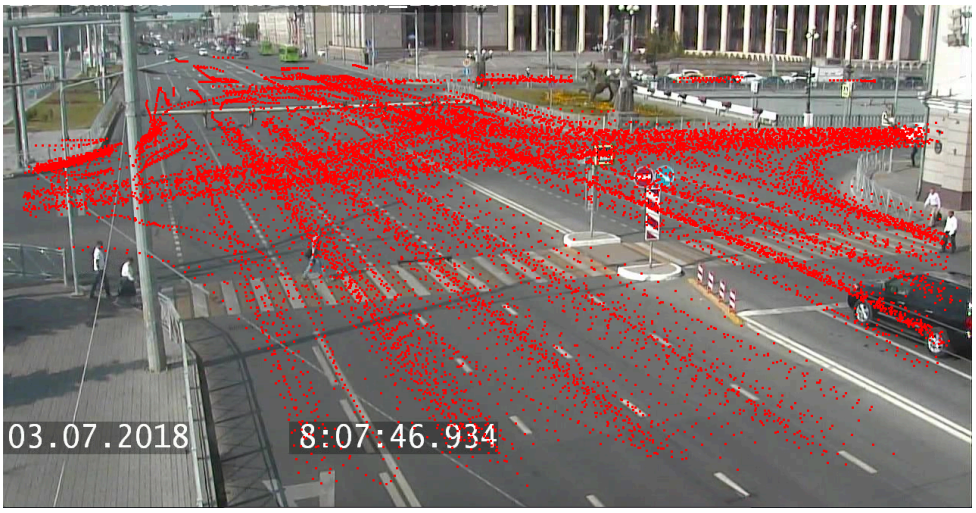


Figure 5.2.2: Output of a tracking system for video the first intersection

5.2.2 Input Data File Structure

Tracking system, as it was described before, handles video from enforcement cameras and prepare it for further analysis: converts video stream into a set of vectors with tracking points on images (Figure 5.2.2).

Input data files have the following structure:

$$[[[(x_1^1, y_1^1), ..., (x_1^n, y_1^n)], [t_1, ...t_n]], [[(x_2^1, y_2^1), ..., (x_2^m, y_2^m)], [t_1, ...t_m]], ...] \tag{5.2.1}$$

As it can be seen from the input data file structure, each trajectory is represented by a two-element array, where first array stores coordinates as an array of two-tuples (x_i^j, y_i^j) and second array contains timestamps for each spatial point in the corresponding order (t_i). The extracted x- and y-coordinates correspond to pixels on input images. In Formula 5.2.1 the lower index of the spatial coordinates indicates the ordering number of a trajectory, while the upper index indicates the ordering number of a tracking point. The outer array refers to the array of trajectories.

5.2.3 Input Data Parsing

Since chosen algorithm requires trajectories in a form of multi-dimensional vectors, the initial input data needs to be converted into the required form. For that reason, a custom parser

was implemented. It takes a 'txt' file with trajectories as an input and as a result it returns a list of Trajectory objects. Trajectory object consists of a number of TrajectoryPoint objects with following information: x-coordinate, y-coordinate, time t. The source code of the parsing method is presented in Appendix A.

As it was mentioned before, the current work is focused on detecting two types of abnormalities: spatial and spatiotemporal. To detect the outliers of the first group it is sufficient to analyze spatial information of trajectories. Detecting outliers of the second group, which is formed by trajectories of vehicles moving with an anomalously low or high speed, requires taking into consideration the temporal information along with spatial. For that reason the average constant speed v is being calculated for each of the input trajectories t at the end of the parsing step using the following equation (Formula 5.2.2):

$$v_{avg}(t) = \frac{distance_{total}}{time_{total}}, \tag{5.2.2}$$

where $distance_{total}$ refers to the total distance between the first and last trajectory points and $time_{total}$ refers to the time elapsed. The total distance can be computed as a sum of Euclidean distances between trajectory points on neighboring frames. Since it is known that frames are taken with an inter-frame interval 0,01 second, the speed calculation can be implemented as follows (Listing 5.1):

Listing 5.1: Speed calculation

```
/**
* Calculates average speed for the trajectory in 'pixels per sec'
*/
public double calcSpeed(Trajectory t) {
  double dist = 0.0;
  for (int i = 0; i < t.length() - 1; i++) {
    dist += t.get(i).distanceTo(t.get(i + 1));
  }

  double time = (t.get(length() - 1).getTime() - t.get(0).getTime()) *
    interFrameTime;

  double avgSpeed = dist / time;
  return avgSpeed;
}

/**
* Calculates the Euclidean distance between two trajectory points
*
* @param this      first (current) trajectory point
* @param other     second trajectory point
* @return          Euclidean distance
```

```
22 */
23 public double distanceTo(TrajectoryPoint other) {
24   if (this == other) {
25     return 0;
26   }
27   double d = Math.pow(this.x - other.x, 2) + Math.pow(this.y - other.y,
       2);
28   return Math.sqrt(d);
29 }
```

5.2.4 Input Data Filtering

Input data contains trajectories of different length and covered distance. However, due to accuracy and tracking errors in tracking system, some trajectories are senseless and look deficient. One of the possible reasons of that is losing the tracking object by a tracker. In contrast with the case of lost location, then the missed location can be found using approximation and regression models, the lost tracking object can not be fixed afterwards. for that reason, in order to improve the quality of results, it was decided to filter the input trajectories and ignore short trajectories with small covered distance, where the covered distance is calculated as an Euclidean distance between first and last trajectory points. For filtering parameters following values were used: $minLength = 10\ (trajectory\ points)$, $minTotalDist = 80\ (pixels)$. Filtering results with depicting the removed (red) and kept (black) trajectories are shown in Figure 5.2.3.

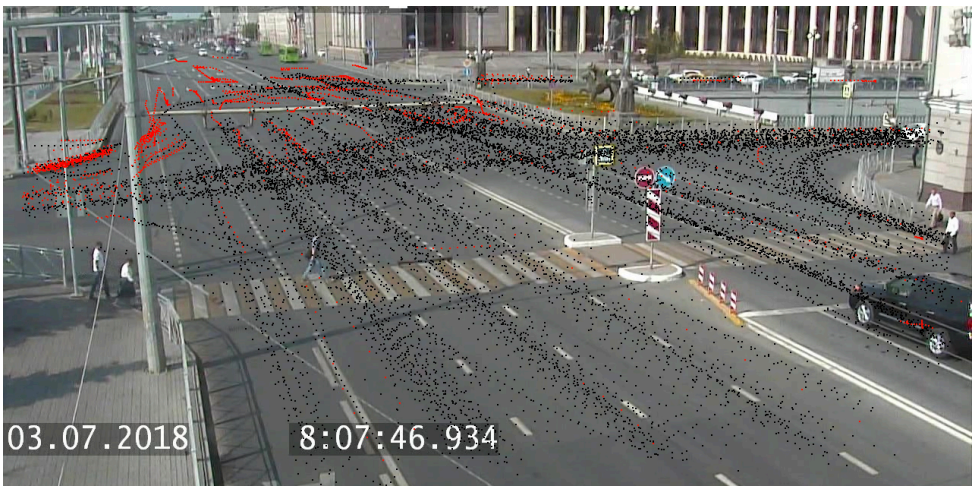


Figure 5.2.3: Results of trajectories filtering for $1.txt$

5.2.5 Trajectories Approximation using Polynomial Regression

As it was discussed before, the polynomial regression will be used to approximate input trajectories. The implementations of a polynomial entity class[1] (needed to further analyze

[1]Polynomial implementation `https://javadoc.io/doc/org.apache.commons/commons-math3/3.4.1/org/apache/commons/math3/analysis/polynomials/PolynomialFunction.html`

the approximation equations and find key points) and a polynomial equation solver[2] from the Apache Commons Math 3.4.1 library were used. To perform a polynomial regression[3] the implementation provided by R. Sedgewick and K. Wayne for Java language was taken [49]. All the ready-to-use implementations were extended by utility methods.

The $PolynomialRegression$ class takes as an input the desired degree of a polynomial (d) and two data sets of N data points consisting of real numbers: array of independent variables ($double[]t$), temporal data in this case, and array of dependent variables ($double[]x$, $double[]y$), spatial x- or y-coordinates. Then it performs a polynomial regression on an input set of N data points (t_i, x_i) or (t_i, y_i) and tries to fit a polynomial $x = \beta_0 + \beta_1 t + \beta_2 t^2 + \ldots \beta_d t^d$, where β_i are the regression coefficients, with an aim to minimize the sum of squared residuals of the multiple regression model. Finding the best solution for polynomial parameters is based on a Least Squares method [44].

In this work the polynomial regression was performed for all the input trajectories (Appendix B). The resulting regression models were compared in terms of R^2 score and analyzed with respect to a trajectory: shape, speed. The following Evaluation Chapter will give the comparison of evaluation results and discuss the obtained results.

Choosing Key Points from Approximated Trajectories

Using the approximated trajectories in further calculation was aimed to decrease the complexity of LCSS calculation. For that reason the length of trajectories must be reduced by choosing several key representative points from the trajectories by analyzing the approximation polynomials.

It is known from Mathematics that critical points of a polynomial $f(t)$ refer to points where the polynomial function is not differentiable or the derivative at that point is equal to zero (stationary points). Stationary points, including local minimum and maximum, rising and falling inflection points, can be found by analyzing the first derivative of a function and solving the $f'(t) = 0$ equation. The inflection points can be found by further analysis of a second derivative: they correspond to the solutions of $f''(t) = 0$ equation. To solve polynomial equations solvers from Apache Commons Math library were used: $LaguerreSolver$, $BisectionSolver$ with following input parameters:

- $maxItem = 30000$ – sets the maximum allowed iterations to find a solution,
- $min = firstTimePoint$ – defines the minimum allowed value for the solution, means the lower border for a solution;

[2]Polynomial Function Solver implementation `https://www.javadoc.io/doc/org.apache.commons/commons-math3/3.4.1/org/apache/commons/math3/analysis/solvers/LaguerreSolver.html`

[3]Polynomial Regression implementation `https://algs4.cs.princeton.edu/14analysis/PolynomialRegression.java`

- $max = lastTimePoint$ – defines the maximum allowed value for the solution, means the upper border for a solution;
- $startValue = min + 1$ – specifies from which value the solver will start searching for a real solution.

The equation solvers were run for first and second derivative polynomial functions taken from polynomials for X- and Y-coordinates. The initiation of a solver is given in Listing 5.2:

Listing 5.2: Polynomial Solver initiation

```
BaseAbstractUnivariateSolver bisectionSolver = new BisectionSolver();
BaseAbstractUnivariateSolver laguerreSolver = new LaguerreSolver();

for (Polynomial derivativeFunc :
      List.of(derivativeFuncX1, derivativeFuncX2,
          derivativeFuncY1, derivativeFuncY2)) {

    for (BaseAbstractUnivariateSolver solver :
          List.of(bisectionSolver, laguerreSolver)) {
      solver.solve(30000, derivativeFunc,
          currentTr.getTrajectoryPoints().stream()
            .mapToInt(TrajectoryPoint::getTime)
            .min().getAsInt(),
          currentTr.getTrajectoryPoints().stream()
            .mapToInt(TrajectoryPoint::getTime)
            .max().getAsInt(),
          currentTr.getTrajectoryPoints().stream()
            .mapToInt(TrajectoryPoint::getTime)
            .min().getAsInt() + 1);
    }
}
```

Solutions found by two solvers are merged together: only points referencing to different to time points are being left. In the case of trajectories analysis inflection points are very significant, because they carry an important information about the shape of a trajectory: such key points can denote the main turns or changes in the trajectory.

However, critical points can not always be found due to computational restrictions of high-order polynomial functions. Consequently, critical points, identified in such a way, can not fully describe the input trajectory and are not sufficient for further analysis since do not provide all the information about borders of a trajectory shape and sometimes can not visualize all the turns. For that reason it was decided to add key points calculated as border points for a trajectory by taking separately minimum and maximum X and Y coordinates and computing the corresponding trajectory points using a respective regression model (Listing 5.3).

Listing 5.3: Calculating the border coordinates and corresponding key points

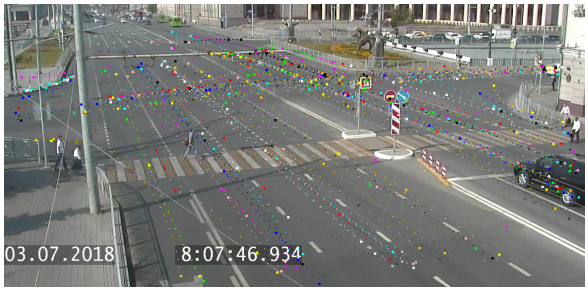
```
// an input image with a resolution 1280*720 -> pixels for X in [0,
   1280], pixels for Y in [0, 720]
double minX = 1280, maxX = 0, minY = 720, maxY = 0;
int tForMinX = 0, tForMaxX = 0, tForMinY = 0, tForMaxY = 0;
for (int time : currentTr.getTrajectoryPoints().stream()
      .mapToInt(TrajectoryPoint::getTime)
      .boxed().collect(toList())) {
  double predictedX = currentTr.getRegressionX()
              .predict(time);
  double predictedY = currentTr.getRegressionY()
              .predict(time);
  if (predictedX < minX) {
    minX = predictedX;
    tForMinX = time;
  }
  if (predictedX > maxX) {
    maxX = predictedX;
    tForMaxX = time;
  }
  if (predictedY < minY) {
    minY = predictedY;
    tForMinY = time;
  }
  if (predictedY > maxY) {
    maxY = predictedY;
    tForMaxY = time;
  }
}

for (int tt : List.of(tForMinX,tForMaxX,tForMinY,tForMaxY)) {
  currentTr.addKeyPoint(new TrajectoryPoint(
    (int) Math.round(currentTr.getRegressionX().predict(tt)),
    (int) Math.round(currentTr.getRegressionY().predict(tt)),
    tt));
}
```

Nevertheless, the total amount of obtained key points is not sufficient to replace the input trajectory for further analysis. At this stage, trajectories in average has following number of key points: $min = 2$, $max = 8$, $average = 3{,}9$.

It is obvious that neither 2 or 4 key points are not enough to describe even the straight trajectory (Figure 5.2.4). Results of clustering using these key points for static and adaptive values of ε are shown on Figure 5.2.5.

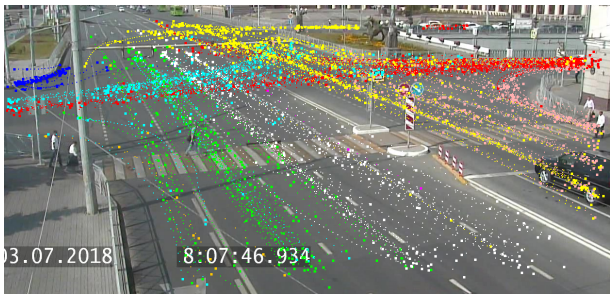
To deal with that and to decrease the influence of the impossibility to find critical points in some cases, obtained key points are augmented with additive key points by choosing them from an input trajectory using following algorithm (Listing 5.4):

(a) trajectories with 2 key points

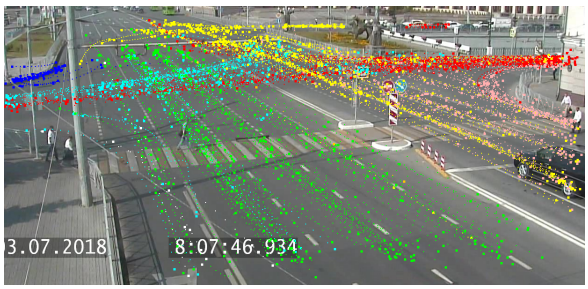


(b) trajectories with 3 key points

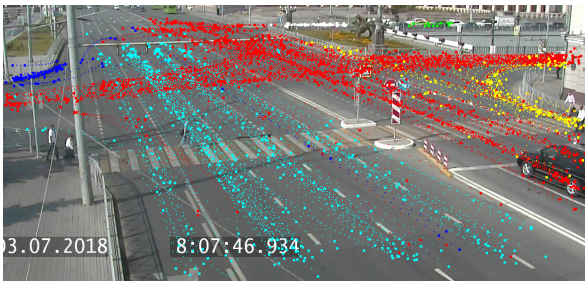
Figure 5.2.4: Trajectories with 2 or 3 key points



(a) 9 clusters, static ε, $coeff_\varepsilon = 0.15$



(b) 8 clusters, static ε, $coeff_\varepsilon = 0.15$



(c) 9 clusters, adaptive ε, $coeff_\varepsilon = 20.0$

Figure 5.2.5: Results of clustering for few key points ($1.txt$)

- define the number of lacking points (to fulfill the total amount of key trajectory points equal to 9 in this case),
- divide the whole trajectory curve into previously calculated number of segments,
- take border points for each of the segments,
- add points, which are neither equal or too close to any of the existing ones.

Listing 5.4: Calculating additive key points

```
1 public static void calcAdditionalKeyPoints(Trajectory currentTr) {
2 //     if small amount of trajectory points
3 //     and trajectory length is more than 2 times bigger than amount of
     key points:
4   if (currentTr.getKeyPoints().size() < MAX_KP_COUNT && currentTr.length
     () >= currentTr.getKeyPoints().size()) {
```

```
    int diff = MAX_KP_COUNT - currentTr.getKeyPoints().size();
    double interval = (currentTr.length() - 3) * 1.0 / diff;
    for (int i = 0; i < diff; i++) {
      int tt = currentTr.get((int) Math.round(1 + interval * i)).getTime
    ();
      Integer bonusTT = (i < diff - 1) ? currentTr.get((int) Math.round
    (1 + interval * (i + 1))).getTime() : null;
      currentTr.addKeyPoint(new TrajectoryPoint(
      (int) Math.round(currentTr.getRegressionX().predict(tt)),
      (int) Math.round(currentTr.getRegressionY().predict(tt)),
      tt), bonusTT);
    }
  }
}
```

So, taken as a whole, the process of choosing key points from approximation polynomials can be summed up as in Listing 5.5.

Listing 5.5: Choosing key points

```
private void calculateKeyPoints(Trajectory currentTr) {
    if (currentTr.length() < MAX_KP_COUNT) {
        currentTr.setKeyPoints(currentTr.getTrajectoryPoints().stream().
    map(TrajectoryPoint::clone).collect(toList()));
        return;
    }

    calculateDiffEquationSolutions(currentTr);
    addFirstAndLastPoints(currentTr);
    calcAdditionalKeyPoints(currentTr);
    sortTrajectoryPoints(currentTr);
}
```

5.2.6 Trajectories Approximation using RDP Algorithm

As well as the Polynomial Regression implementation, for traditional RDP approximation approach the implementation by Lukasz Wiktor[4] was used. The method takes the original trajectory points and positive $'varepsilon$ parameter value as inputs. The method returns decreased amount of original trajectory points. The algorithms works in a recursive way by iteratively dividing the input list of trajectory points and working with two parts. Since on each step algorithms searches for the point farther than the ε from the closest simplification line till no such points left, the resulting amount of trajectory points in the so-called simplified trajectory can be almost the same as in the initial one in the case of complex trajectories.

[4]Ramer-Douglas-Peucker implementation `https://github.com/LukaszWiktor/series-reducer`

5.2.7 Trajectories Approximation using Douglas-Peucker N Algorithm

The implementation of Douglas-Peucker N algorithm was written from scratch based on the Algorithm 4 and is given in Listing 5.6.

Listing 5.6: Douglas-Peucker N implementation

```
/**
 * Reduces number of points in given series using Douglas-Peucker N
   algorithm
 * to the total amount of given 'count'.
 *
 * @param points    initial, ordered list of points {@link
   TrajectoryPoint}
 * @param count     allowed margin of the resulting curve, has to be > 0
 */
public static List<TrajectoryPoint> reduceToN(List<TrajectoryPoint>
   points, int count) {
    List<TrajectoryPoint> pointsCopy = points.stream()
            .map(TrajectoryPoint::clone).collect(toList())
            .subList(1, points.size() - 1);
    if (count < 2) {
        throw new IllegalArgumentException("Points count cannot be less
   then 2.");
    }
    double furthestPointDistance = 0.0;
    TrajectoryPoint furthestPoint = null;
    List<TrajectoryPoint> simplified = new ArrayList<>();
    simplified.add(points.get(0));
    simplified.add(points.get(points.size() - 1));
    while (simplified.size() < count) {
//        for each original point from pointsCopy -> define line segment
   , calc the distance
        for (TrajectoryPoint point: pointsCopy) {
            double dist = calcDist(point, simplified);
            if (dist > furthestPointDistance) {
                furthestPointDistance = dist;
                furthestPoint = point;
            }
        }
        if (furthestPoint == null)
            break;
        simplified.add(furthestPoint);
        pointsCopy.remove(furthestPoint);
        furthestPointDistance = 0.0;
        furthestPoint = null;
    }
```

```
36      return simplified;
37  }
```

Algorithm does not work with trajectories containing less than 2 *TPs*, since at least first and last points from the original curve must be kept in the simplified trajectory. Method requires the desired length of approximated trajectory $count$ as input parameter and iteratively chooses the farthest point. The stop condition is defined by $count$ parameter.

In comparison with the Polynomial Regression method, the accuracy of the RDP algorithm can be managed by changing the maximum distance ε for the traditional RDP and by changing the N value for the Douglas-Peucker N.

5.3 Trajectories Analysis

5.3.1 Similarity Measure Calculation

As it was mentioned before, LCSS measure will be used as a similarity measure. Consequently, LCSS distance will be calculated based on a LCSS similarity according to above mentioned formulas. It is worth noting that LCSS distance is symmetric and for pair of trajectories can be computed just once [30].

Notwithstanding that the implementation of LCSS similarity measure exists in R package [32], it does not allow δ and ε parameters to be dynamic. For that reason the custom implementation was written. The method for LCSS calculation is presented in Listing 5.7.

Listing 5.7: LCSS calculation

```
1  /**
2  * Calculates LCSS for two input trajectories
3  *
4  * @param t1      first trajectory
5  * @param t2      second trajectory
6  * @param δ       δ parameter: how far we can look in time to match a given
       point from one T to a point in another T
7  * @param ε       ε parameter: the size of proximity in which to look for
       matches
8  * @return        LCSS for t1 and t2
9  */
10 private Double calcLCSS(Trajectory t1, Trajectory t2, Double δ, Double
       εx, Double εy) {
11   int m = t1.length();
12   int n = t2.length();
13
14   if (m == 0 || n == 0) {
15     return 0.0;
16   }
17
```

```
18 //        calculate adaptive ε for X and Y
19   if (IS_ADAPTIVE) {
20     TrajectoryPoint tp1 = t1.getKeyPoints().get(m - 1);
21     TrajectoryPoint tp2 = t2.getKeyPoints().get(n - 1);
22     epsilonX = getEpsilonX(tp1, tp2, LinkageMethod.AVERAGE);
23     epsilonY = getEpsilonY(tp1, tp2, LinkageMethod.AVERAGE);
24   }
25
26 //      check last trajectory point (of each trajectory-part recursively
       )
27 //      according to [8]: delta and epsilon as thresholds for X- and Y-
       axes respectively
28 //      Then the abscissa difference and ordinate difference are less
       than thresholds (they are relatively close to each other)
29 //      they are considered similar and LCSS distance is increased by 1
30   if (abs(t1.get(m - 1).getX() - t2.get(n - 1).getX()) < εx
31       && abs(t1.get(m - 1).getY() - t2.get(n - 1).getY()) < εy
32       && abs(m - n) <= δ) {
33     return 1 + calcLCSS(head(t1), head(t2), δ, ε);
34   } else {
35     return max(
36       calcLCSS(head(t1), t2, δ, εx, Double εy),
37       calcLCSS(t1, head(t2), δ, εx, Double εy));
38   }
39 }
40
41 /**
42 * Calculates shortened trajectory by excluding last trajectory point
43 *
44 * @param t trajectory
45 * @return trajectory without last trajectory point
46 */
47 private Trajectory head(Trajectory t) {
48   Trajectory tClone = t.clone();
49   tClone.getTrajectoryPoints().remove(tClone.length() - 1);
50   return tClone;
51 }
```

5.3.2 Clustering

Agglomerative Hierarchical Clustering

In accordance with the general workflow of the framework, the clustering is being initiated after the completion of parsing, filtering and approximation of the original trajectories and calculating the LCSS distance for each pair of trajectories (Listing 5.8).

Listing 5.8: Clustering initiation

```
// parsing
List<Trajectory> trajectories = parseTrajectories(inputFileName);

// filtering
trajectories = trajectories.stream().filter(tr ->
    tr.length() > MIN_LENGTH && tr.totalDist() >= MIN_TOTAL_DIST)
  .collect(toList());

// approximation
performRegression(trajectories, input);

// calculate LCSS distances
Double[][] trajLCSSDistances = calcLCSSDistances(trajectories);

// clustering
List<Cluster> clusters = new Clustering().cluster(trajectories);

// clusters visualization
displayClusters(inputImgFileName, clusters);
```

Since no appropriate implementation of hierarchical clustering for trajectories with the use of LCSS distance and capable of taking an adaptable parameters values were found, the clustering as well as LCSS similarity calculation was written from scratch guided by Algorithm 5 outlined above. The source code of clustering is given in Appendix C.

Clustering is done in an iterative way of joining two closest clusters into one with following recalculation of a clusters similarity (proximity) matrix. The clustering method takes as an input the $OUTPUT\_CLUSTERS\_COUNT$ parameter which controls when the clustering will stop and defines the total amount of resulting clusters. If no value is passed, it will be considered as 1 and the clustering will be done till all clusters are merged into one in concordance with the basic algorithm of agglomerative hierarchical clustering, or till the further joining becomes infeasible.

DBSCAN Clustering

To perform DBSCAN clustering the implementation[5] provided by the Apache Commons Math 3.4.1 library was used. Since method requires the data in a form of points, which can be vectors of equal length, the trajectory data needs to be preprocessed (Listing 5.9).

Listing 5.9: Trajectories flattening

```
/**
 * Flatten all the given trajectories by representing each trajectory as
    a list of points:
```

[5]DBSCAN Clustering implementation https://www.javadoc.io/doc/org.apache.commons/commons-math3/3.4.1/org/apache/commons/math3/ml/clustering/DBSCANClusterer.html

```
 * {(x1, y1), (x2, y2), ... (xn, yn)} --> {x1, y1, x2, y2, ..., xn, yn}
 * All the input trajectories must be of the same length
 *
 * @param trajectories    list of input trajectories
 */
public static void flattenTrajectories(List<Trajectory> trajectories) {
    boolean diffLength = trajectories.stream().anyMatch(tr ->
    getTrajectoryPoints(tr).size() != MAX_KP_COUNT);
    if (diffLength) {
        throw new UnsupportedOperationException("DBSCAN can not be
    applied to vectors of different length");
    }
    for (Trajectory trajectory : trajectories) {
        List<TrajectoryPoint> keyPoints = getTrajectoryPoints(trajectory
    );
        double[] tps = new double[2 * keyPoints.size()];
        for (int i = 0; i < keyPoints.size(); i++) {
            tps[2 * i] = keyPoints.get(i).getX();
            tps[2 * i + 1] = keyPoints.get(i).getY();
        }
        trajectory.setPoint(tps);
    }
}
```

Clusters validation

As described earlier, cluster validation will be done using the Dunn Index (DI). Listing 5.10 provides an implementation of the DI value calculation for the resulting clusters.

Listing 5.10: DI calculation

```
/**
 * Dunn's Validity Index (DI) = dist_min / diam_max
 * dist_min = min inter-cluster distance (minimum distance between two
    clusters;
 * single-linkage -> min distance between two trajectories from wo
    clusters)
 * diam_max = max intra-cluster distance (maximum distance between two
    farthermost trajectories)
 */
private void validateClusters() {
  // sort trajectories inside each cluster according to trajectory ID
    clusters.forEach(cluster ->
        cluster.getTrajectories().sort(Comparator.comparing(Trajectory::
    getId)));

    double minDist = Double.MAX_VALUE;
```

```
13      for (int i = 0; i < clusters.size(); i++) {
14          for (int j = i + 1; j < clusters.size(); j++) {
15              if (clustLCSSDistances[clusters.get(i).getId()][clusters.get
    (j).getId()] < minDist)
16                  minDist = clustLCSSDistances[clusters.get(i).getId()][
    clusters.get(j).getId()];
17          }
18      }
19
20      double maxDiam = clusters.stream().mapToDouble(cluster -> {
21          double maxDist = 0;
22          for (int i = 0; i < cluster.getTrajectories().size(); i++) {
23              for (int j = i + 1; j < cluster.getTrajectories().size(); j
    ++) {
24                  if (trajLCSSDistances[cluster.getTrajectories().get(i).
    getId()][cluster.getTrajectories().get(j).getId()] > maxDist)
25                      maxDist = trajLCSSDistances[cluster.getTrajectories
    ().get(i).getId()][cluster.getTrajectories().get(j).getId()];
26              }
27          }
28          return maxDist;
29      }).max().getAsDouble();
30
31      double DI = minDist / maxDiam;
32      LOGGER.info(String.format("DI = %.2f", DI));
33  }
```

As it can be seen from the presented code listing, the DI calculation consists of two parts: first calculation of the minimum distance between all pairs of clusters ($minDist$), then computing the maximum diameter among the diameters of all clusters ($maxDiam$). Since the implementation is based on the assumption that trajectories in the clusters appear in ascending order of identifiers and in view of this the inner loop starts with an index that is 1 more than the external index of the cluster or trajectory, to simplify further calculations of DI the cluster trajectories will be ordered beforehand.

5.3.3 Clusters' Modeling

To implement cluster modeling, the $Cluster$ entity class was augmented by the $clusterModel$ attribute of the $Trajectory$ type which stores a representative trajectory for the current cluster. Listing 5.11 presents an implementation of the *CM* calculation. For each cluster, the trajectory with the smallest average LCSS distance to all other trajectories in the current cluster is being specified. This trajectory is chosen as a *CM*. In the case of clusters consisting of a single trajectory, this trajectory is considered to be a *CM*.

Since cluster's model defines the most representative trajectory for the cluster, it simplifies the classification process, because it is not needed to calculate the distances from an input trajectory to all the trajectories from each cluster one by one. The *CM* acts like a cluster.

Listing 5.11: Clusters' modeling

```
private void clustersModeling() {
  for (Cluster c: clusters) {
    if (c.getTrajectories().size() == 1) {
      c.setClusterModel(c.getTrajectories().get(0));
      continue;
    }
    Trajectory model = null;
    double avg = Double.MAX_VALUE;
    for (Trajectory t: c.getTrajectories()) {
      double sum = 0.0;
      for (Trajectory t1: c.getTrajectories()) {
        if (t1 != t)
        sum += trajLCSSDistances[t.getId()][t1.getId()];
      }
      double curAvg = sum / (c.getTrajectories().size() - 1);
      if (curAvg < avg) {
        model = t;
        avg = curAvg;
      }
    }
    c.setClusterModel(model);
  }
}
```

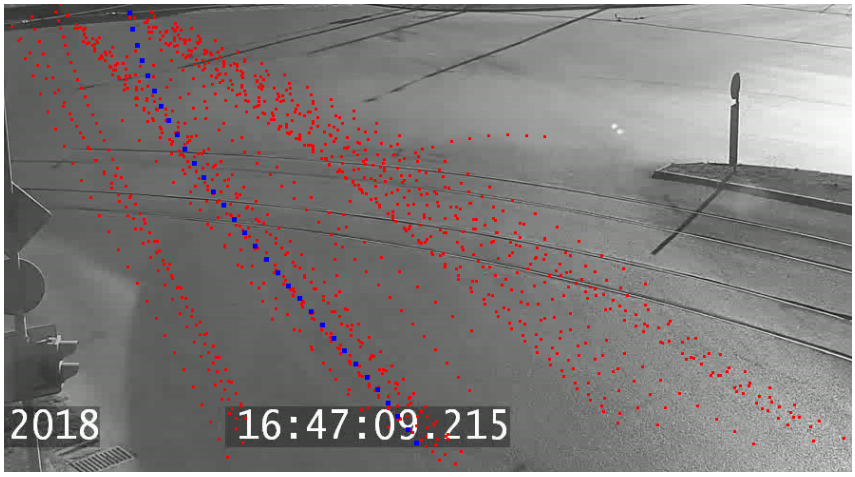
The most representative cluster modeling results are shown in Figure 5.3.1. All trajectories belonging to the cluster are depicted using red dots, *CMs* are highlighted in blue.

5.3.4 Trajectories Classification and Anomalies Detection

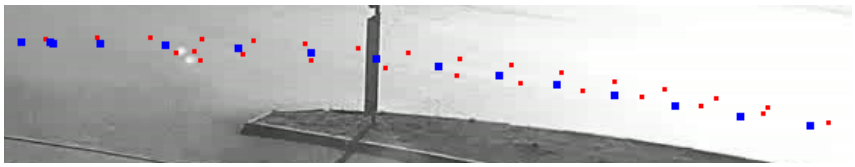
To perform anomalies detection, trajectories classification is used, which is based on analyzing *CMs* and finding the closest one or defining the input trajectory as anomalous. Method calculates distances from the input trajectory to each of the existing *CMs*. A threshold $lcssMax$ is used to define the minimum required similarity between trajectory and a cluster to be considered belonging. It is required to identify the anomalous trajectories, significantly different from known *CMs*. The whole method of a trajectory classification is given in Listing 5.12:

Listing 5.12: Trajectories classification

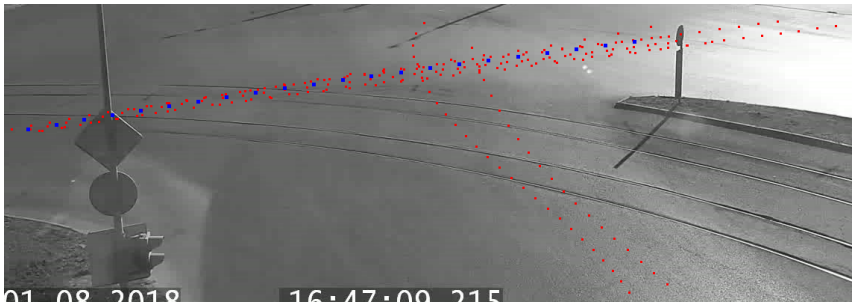
```
    /**
     * Trajectories classification
```



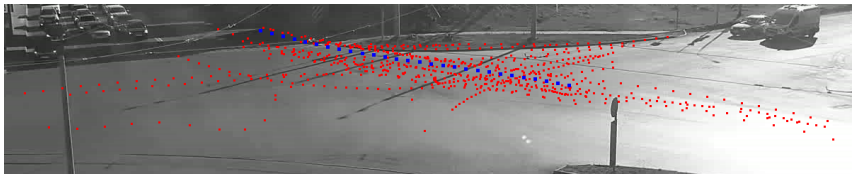
(a) cluster 1



(b) cluster 3



(c) cluster 4



(d) cluster 7

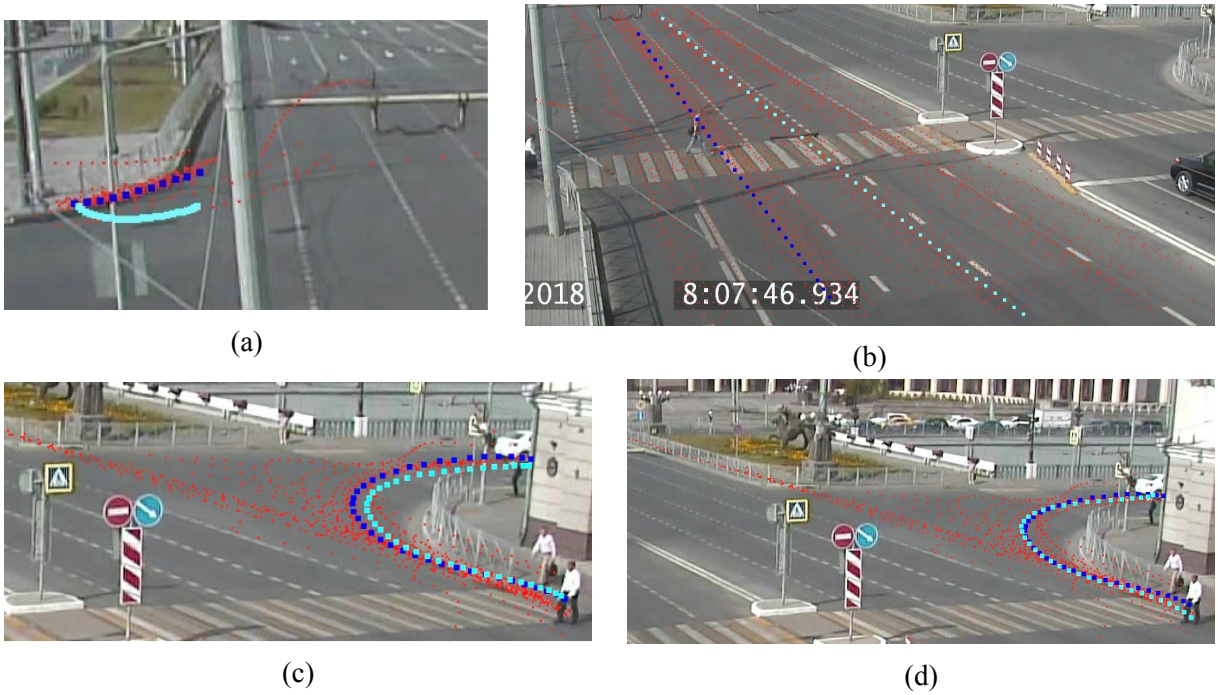
Figure 5.3.1: Results of clusters' modeling ($2.txt$)

```
3     * Based on using defined clusters' models
4     * - compares input trajectory with each cluster by calculating the
    LCSS distance to the cluster model
5     * - finds the closest cluster taking into consideration threshold
    value *defining the maximum allowed distance)
6     * - - no cluster was found -> trajectory is anomalous (unknown
    behavior)
7     * - - cluster is found -> depends on the label of the cluster
8     *
```

```
9       * @param inputTrajectories A list of input trajectories to be
    classified
10      *
11      * Classifies each input trajectory and print the classification
    result
12      */
13     public void classifyTrajectories(List<Trajectory> inputTrajectories)
     throws IOException {
14         double lcssMax = 0.85;
15         List<Trajectory> anomalousTrajectories = new ArrayList<>();
16         inputTrajectories.forEach(it -> {
17             final double[] minLcss = {1.0};
18             final Cluster[] closestCluster = {null};
19             calcEuclDistancesToCP(inputTrajectories);
20
21             System.out.println(String.format("------tr \%s-----", it.
    getId()));
22             clusters.forEach(cl -> {
23                 double curLcss = calcLCSSDist(it, cl.getClusterModel());
24                 if (curLcss < minLcss[0]) {
25                     minLcss[0] = curLcss;
26                     closestCluster[0] = cl;
27                 }
28                 System.out.println(String.format("dist to cl \%s = \%.2f
    ", cl.getId(), curLcss));
29             });
30             if (closestCluster[0] == null || minLcss[0] > lcssMax) {
31                 System.out.println("anomalous trajectory");
32                 anomalousTrajectories.add(it);
33             } else {
34                 System.out.println(String.format("closest cl is \%s",
    closestCluster[0].getId()));
35                 System.out.println(closestCluster[0].getNormal() ? "
    normal trajectory" : "anomalous trajectory");
36             }
37         });
38         new DisplayImage().displayAndSave(getImgFileName("1"), null,
    null, anomalousTrajectories, false);
39     }
```

Classification algorithm prints out the classification results with specifying the closest clusters with the calculated similarity value.

Figures 5.3.2 – 5.3.3 present the classification results using normal trajectories. Red trajectories denote the closest cluster; blue trajectory, depicted using bold *TPs*, emphasizes the corresponding *CM*; the input trajectory plotted using cyan color.

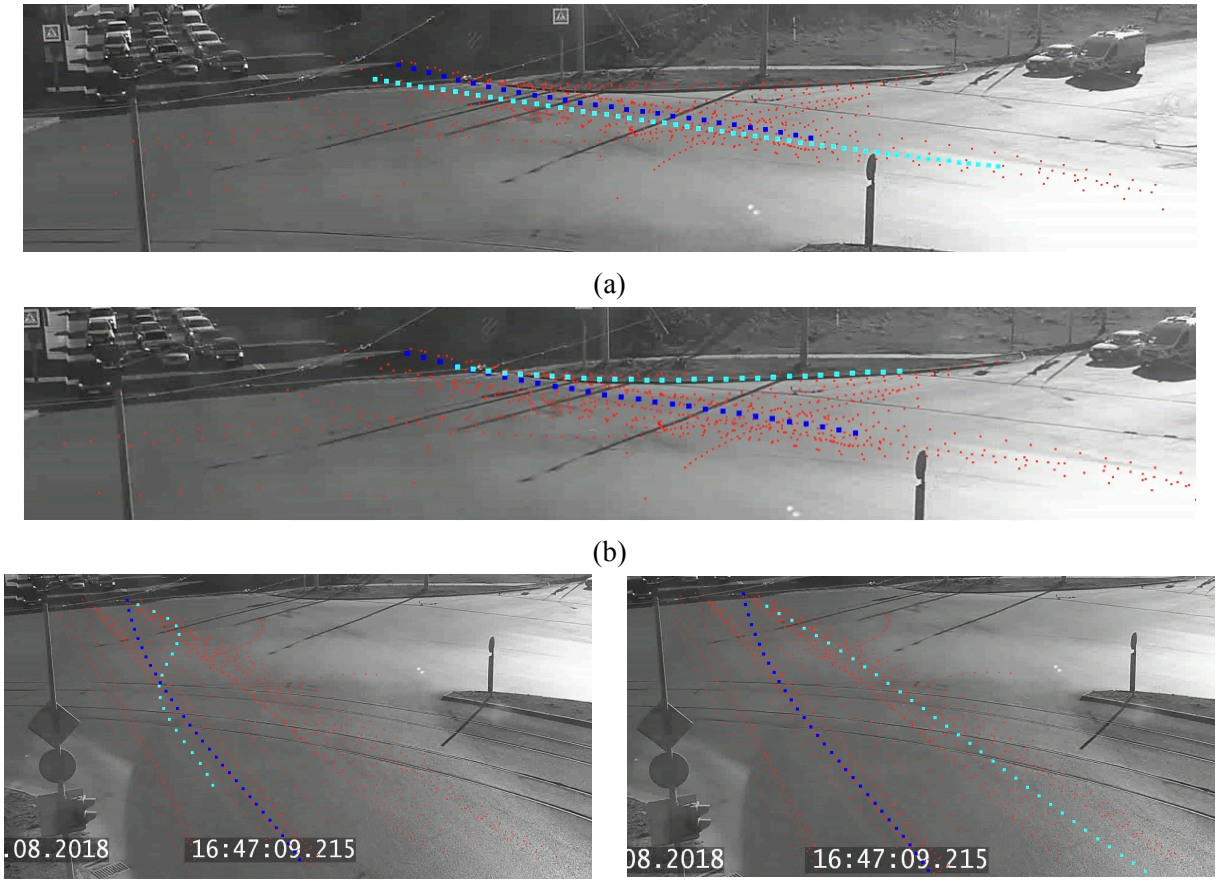


(a) (b)

(c) (d)

Figure 5.3.2: Results of normal trajectories classification ($1.txt$)

Results of anomalous trajectories clustering are given along with the calculated similarity, denoting the belonging of trajectory to the cluster (Figure 5.3.4).



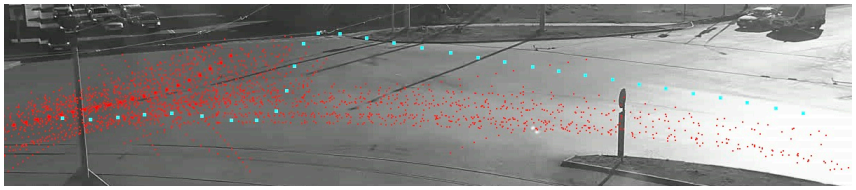
(a)

(b)

(c)

(d)

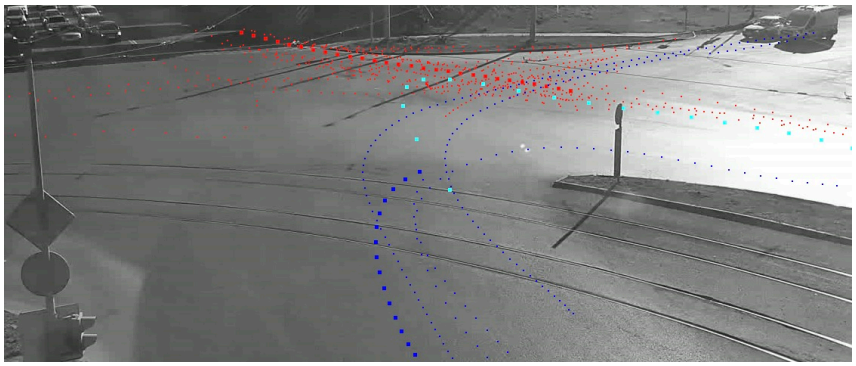
Figure 5.3.3: Results of normal trajectories classification ($2.txt$)



(a)

```
------tr 51-----
dist to cl 0 = 1.00
dist to cl 10 = 0.88
dist to cl 25 = 1.00
dist to cl 26 = 1.00
dist to cl 33 = 1.00
dist to cl 48 = 1.00
dist to cl 49 = 1.00
dist to cl 190 = 1.00
anomalous trajectory
```

(b)



(c)

```
------tr 51-----
dist to cl 0 = 1.00
dist to cl 10 = 0.88
dist to cl 25 = 1.00
dist to cl 26 = 1.00
dist to cl 33 = 1.00
dist to cl 48 = 1.00
dist to cl 49 = 1.00
dist to cl 190 = 1.00
anomalous trajectory
```

(d)

Figure 5.3.4: Results of anomalous trajectories classification ($2.txt$)

6 Results & Evaluation

This chapter describes the evaluation tests performed on the proposed approach on the basis of developed framework. Following sections are intended to give information about input data preparation, present and discuss obtained results. To start with, the trajectories approximation using polynomial regression is considered. After that the accuracy of clustering step is tested using an aforementioned DI index. As a conclusion, output results of a classification step are discussed, validation of the anomalies detection quality is done.

6.1 Evaluation of Correctness and Accuracy

This section gives details about evaluation tests performed to validate the accuracy of obtained results.

6.1.1 Results of Trajectories Approximation using Polynomial Regression

To make a decision about degrees of approximation polynomials, several experiments were run trying to fit the input trajectories into polynomials of 3rd, 4th, 5th degrees respectively. The following table depicts minimum and average values of R^2 metric for each of the experiments (Table 6.1.1).

This paragraph will discuss results of experiments using trajectories from first intersection ($1.txt$) before and after filtering the trajectories. It can be seen from the Table 6.1.1 that the average values of R^2 score are acceptable for all the experiments. However, the minimum R^2 values, which are equal to 0,66 and 0,466 for the first experiment with only 3rd-degree polynomials for unfiltered trajectories, are poor and unsatisfactory, meaning that model can predict barely half of the points for some trajectories correctly. Filtering out trajectories with short displacements in terms of both space and time improved the results significantly. However, results for approximation using 3rd-degree polynomials are still insufficient. This can affect following analysis and worsen further analysis. For that reason approximation using several degrees was performed in a following way:

1. firstly perform approximation using the lowest degree of a polynomial as a starting point,
2. compare the obtained R^2 with a predefined threshold (0,98 in this case); if the obtained value is less than the threshold value, increase the degree and reperform polynomial regression,
3. continue till the acceptable R^2 is obtained or till reaching the limit for a polynomial degree to check (5 in this case).

Approximation using 3rd and 4th degree polynomials in conjunction improved both minimum and average values of R^2 drastically. Though adding a 5th degree polynomial into consideration

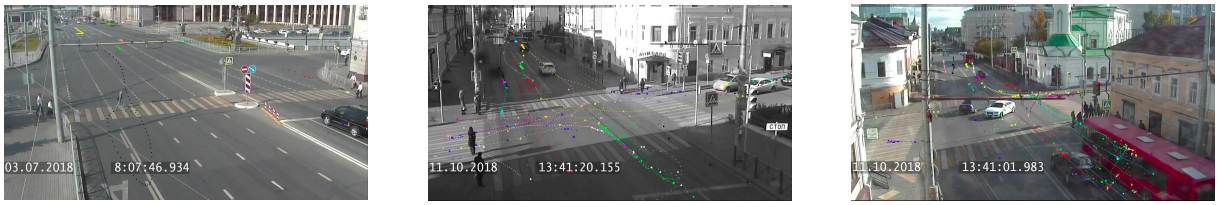
Table 6.1.1: R^2 values for different degrees of polynomials

Degrees of polynomials	R^2 score			
	X		Y	
	min	avg	min	avg
$1.txt$ (before filtering)				
{3}	0.66	0.994	0.466	0.989
{3, 4}	0.897	0.998	0.823	0.994
{3, 4, 5}	0.949	0.998	0.864	0.995
$1.txt$				
{3}	0.689	0.997	0.777	0.995
{3, 4}	0.942	0.999	0.872	0.997
{3, 4, 5}	0.98	0.999	0.88	0.997
$2.txt$				
{3, 4}	0.992	0.9997	0.832	0.996
$3.txt$				
{3, 4}	0.815	0.995	0.867	0.996
$4.txt$				
{3, 4}	0.879	0.995	0.722	0.993

did not affect results significantly and improved the average coefficient only for 0,01 in comparison with the previous experiment. In view of this it was decided to focus on approximation using 3rd and 4th degrees and run the experiments on rest of input trajectories data sets ($2.txt$, $3.txt$, $4.txt$).

For the sake of simplicity the trajectories are classified into two groups depending on a degree of polynomials used to approximate: first group contains trajectories approximated with a 3rd degree polynomial functions while the second group consists of trajectories approximated with a 4th degree polynomial functions. Both groups were analyzed in terms of a shape and an average speed. Figure 6.1.1 depicts second group of trajectories and Table 6.1.2 gives the minimum, average and maximum speed of trajectories for both groups.

Thereinafter the trajectories approximated with 3rd- and 4th-degree polynomial functions will be referred to as a first and second groups of trajectories respectively. It can be observed that trajectories different groups have the widely differing speeds. The first group includes trajectories with much higher speeds, particularly striking is that the maximum speed for the



(a) $1.txt$ (b) $3.txt$ (c) $4.txt$

Figure 6.1.1: Trajectories approximated with 4^{th}-degree polynomial functions

Table 6.1.2: Overview of min, avg and max speeds of vehicles

Degree of a polynomial	Speed *(pixels per sec)*		
	min	avg	max
$1.txt$ (before filtering)			
{3}	18.555	335.365	1721.499
{4}	1.206	72.34	374.396
$1.txt$			
{3}	61.814	372.435	909.121
{4}	26.603	229.053	602.773
$2.txt$			
{3}	85.705	494.016	1107.96
{4}	183.087	613.865	900.737
$3.txt$			
{3}	13.65	301.481	1012.748
{4}	29.26	206.119	764.25
$4.txt$			
{3}	43.01	269.074	872.33
{4}	22.92	163.431	708.154

second group is almost equal to average speed for the first group and the average speed for the first group is almost four times as much as for the second group for the case of unfiltered trajectories. After excluding the short trajectories from the consideration results became more smooth, however the same observation still takes place and is still noticeable. Also the picture depicts that 4^{th}-degree polynomial functions were used to approximate trajectories of complex shape or trajectories with densely located trajectory points.

Such a tendency is mostly distinguishable while considering trajectories from the first intersection, which can supposed to be the representative set of trajectories since it has the largest amount of data instances. It is also worth noting that the second trajectories data set ($2.txt$) has only 10 trajectories in the first group, hence the trajectories speed analysis can not be very accurate.

The results of performed polynomial regression approximation and key points calculation are presented in Figure 6.1.2. The original trajectory data is visualized using a red color, while the blue trajectory points correspond to data points obtained using the functions of a polynomial approximation for the same time points. Key points of each trajectory are emphasized using bold blue square dots. It can be seen that the approximation function is close to the original trajectory function, for some trajectories approximation gives the same coordinates as in the original data (for trajectories with approximation polynomials having $R^2 \approx 1.0$). Also key points accurately follow the approximation line and describe the trajectory properly, hence, they can be used instead of original trajectory points to simplify the further trajectories analysis.

Summary

Hence, it follows up that the higher-order polynomial functions are preferred to approximate trajectories of following groups:

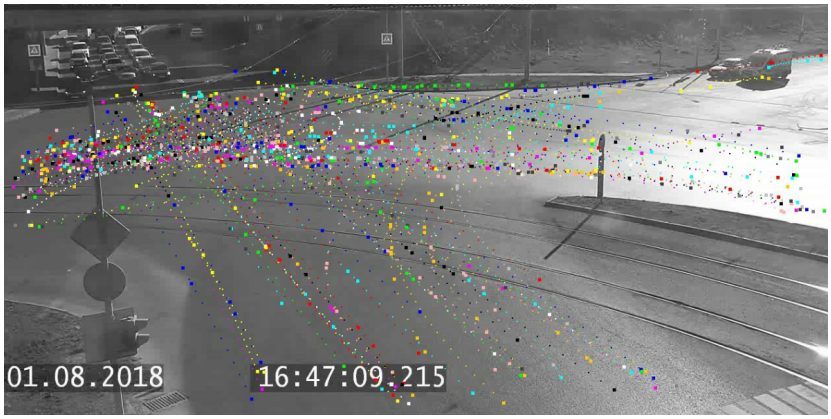
- slow-moving or inactive vehicle trajectories (including trajectories of vehicles waiting at the intersections), detectable on Figure 6.1.1a;
- vehicle trajectories with a non-constant speed or an acceleration on some sections of a path (can be represented by non-equal allocation of trajectory points, where dense areas of trajectory points signalize about acceleration during these time intervals; as a result low-order polynomial can not describe such a complex dependency between a coordinate and a time);
- trajectories of complex shapes (sharp turns, Pascal snails), especially recognizable on Figures 6.1.1b-c.

6.1.2 Results of Trajectories Approximation using RDP Algorithm

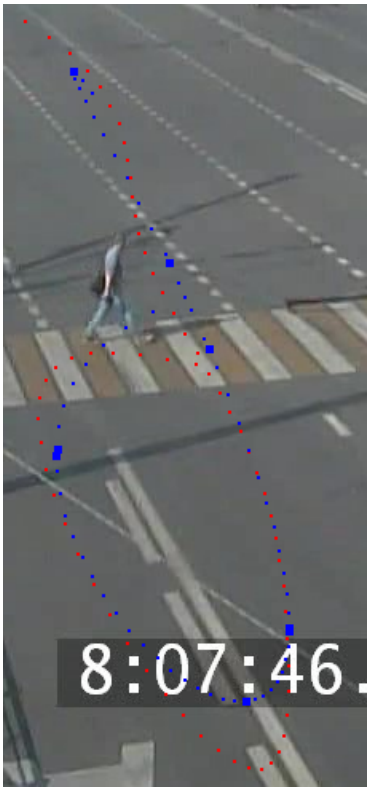
Traditional RDP Algorithm

Comparison of total length of approximated trajectories and the positional errors are presented in Table 6.1.3. As it can be seen, starting from the $\varepsilon = 2.0$ approximation results are not usable. For the case of $\varepsilon = 2.0$ the average length of approximated trajectory is equal to the desired length, however many trajectories remain too long.

However, notwithstanding that traditional RDP algorithm chooses adequate, representative points, it does not guarantee the exact amount of resulting points. Moreover, for trajectories of vehicles moving in a straightforward direction, meaning all original *TPs* are on a straight line,



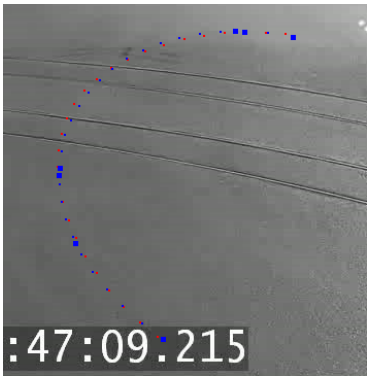
(a) full 2.txt



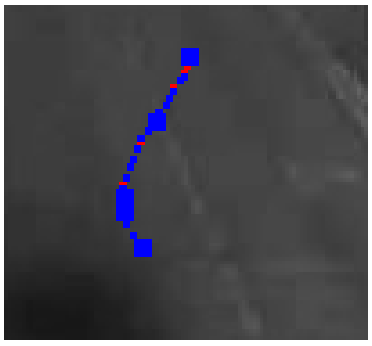
(b) loop from 1.txt

(c) turn from 1.txt

(d) complex 1.txt



(e) curve from 2.txt



(f) complex from 3.txt

(g) curve from 3.txt

(h) curve from 4.txt

Figure 6.1.2: Results of polynomial regression with emphasizing key points

Table 6.1.3: Overview of min, avg and max length and positional errors for trajectories, approximated with RDP algorithm

Length *(TPs)*			Positional Error *(pixels)*		
min	avg	max	min	avg	max
$\varepsilon = 0.5$					
2	11	43	0.0	3.1	11.03
$\varepsilon = 1.0$					
2	6.4	33	0.0	7.2	30.68
$\varepsilon = 1.5$					
2	5.35	31	0.0	10.26	39.24
$\varepsilon = 2.0$					
2	4.7	24	0.0	14.1	61.3
$\varepsilon = 10.0$					
2	2.64	13	0.0	72.117	354.62
$\varepsilon = 10.5$					
2	2.6	12	0.0	77.66	505.73
$\varepsilon = 11.0$					
2	2.24	7	0.0	137.31	1035.12

RDP algorithm will keep only first and last points. Such a behavior describes the trajectory form accurately, but do not appropriate to use for clustering. That leads to the necessity to perform a completion of approximation points by calculating additional key points.

Douglas-Peucker N Algorithm

The same comparison is performed for the Douglas-Peucker N Algorithm for different values of N, results are given in Table 6.1.4. The minimum length is equal to 2, meaning that *TPs* are located on a straight line, and the trajectory coincides with the initial simplifying line, consisting of the first and last trajectory points (Figure 6.1.4a). Notwithstanding that, the average length is almost equal to the desired N value. Moreover, as expected, increase of N value leads to the decrease of the positional error.

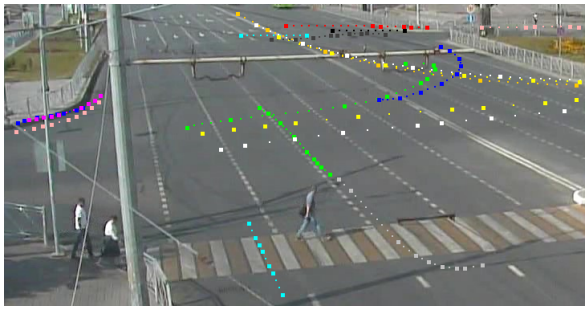
Simplified trajectories obtained by applying the Douglas-Peucker N algorithm are presented in Figure 6.1.3 for the parameter value $N = 9$, representing the desired amount of simplification points.

(a) curve from 1.*txt*

(b) turn from 1.*txt*

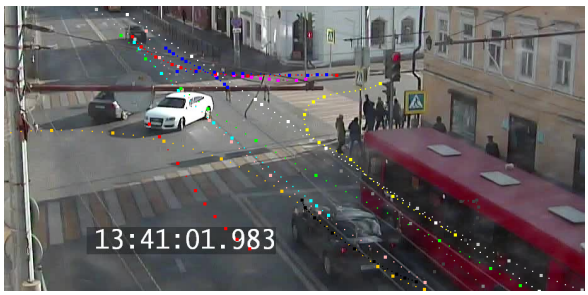
(c) loop from 1.*txt*

(d) straight trajectories 1.*txt*



(e) turns from 1.*txt*

(f) selection from 3.*txt*



(g) selection from 4.*txt*

Figure 6.1.3: Results of Douglas-Peucker N simplification

Trajectories approximated by only first and last points will be complemented by calculating middle points to fulfill the desired maximum amount of *TPs* in a simplified trajectory (Figure 6.1.4b). Some of the points are redundant, but improve the clustering results.

(a) before adding middle points

(b) complemented by middle points

Figure 6.1.4: Douglas-Peucker N 3.txt

Table 6.1.4: Overview of min, avg and max length and positional errors for trajectories, approximated with Douglas-Peucker N algorithm

Length *(TPs)*			Positional Error *(pixels)*		
min	avg	max	min	avg	max
$N = 7$					
2	6.86	7	0.0	12.2	472.36
$N = 8$					
2	7.82	8	0.0	9.26	388.85
$N = 9$					
2	8.76	9	0.0	7.38	340.56

Summary

In order to choose between aforementioned implemented approximation methods, the obtained results need to be compared using the same accuracy evaluation method. For that the Positional errors metric was chosen. It can work with both algorithms, because Polynomial Regression, as well as Douglas-Peucker N, simplifies the input trajectory into a reduced number of key points. Positional error value demonstrates how far is the obtained simplified curve from the original.

Results of comparison are given in Table 6.1.5. For trajectories approximated using a Polynomial Regression positional errors are calculated on the set of final key points, consisting of the solutions of derivative equations, border and middle points.

As it can be seen, Douglas-Peucker N algorithm leads to much more accurate results with smaller positional errors and desired average length of simplified trajectories for all the input files with trajectories.

Table 6.1.5: Comparison of min, avg, max lengths and positional errors for trajectories, simplified by Polynomial Regression and Douglas-Peucker N algorithms

Length *(TPs)*			Positional Error *(pixels)*		
min	avg	max	min	avg	max
$1.txt$					
Polynomial Regression, $N = 9$					
6	8.34	9	0.0	29.18	959.48
Douglas-Peucker N, $N = 9$					
7	8.97	9	0.0	7.26	340.56
$2.txt$					
Polynomial Regression, $N = 9$					
6	8.18	9	0.1	21.01	870.94
Douglas-Peucker N, $N = 9$					
7	8.97	9	0.0	4.9	92.82
$3.txt$					
Polynomial Regression, $N = 9$					
6	8.46	9	0.74	91.32	2015.24
Douglas-Peucker N, $N = 9$					
8	8.99	9	0.0	20.36	258.09
$4.txt$					
Polynomial Regression, $N = 9$					
7	8.42	9	0.87	74.10	1423.47
Douglas-Peucker N, $N = 9$					
8	8.98	9	0.0	14.11	299.45

6.1.3 Trajectories Clustering Results

To compare the clustering results, several experiments were carried out on different input data ($1.txt$, $2.txt$), using static and adaptive values of the ε parameter, with a different number of final clusters (7, 8, 9). The obtained clusters were validated in two ways: visual test and DI value. The results of experiments and discussion will be presented below.

To evaluate the efficiency of using the adaptive values of the ε parameter, and compare them with the results when using static values, clustering was performed on the data from the

first intersection ($1.txt$) for 7, 8 and 9 resulting clusters at different values of the coefficient participating in the formulas for calculating ε (6.1.1-6.1.5):

$static : coeff = \{\mathbf{0.1}, \mathbf{0.15}, \ldots\}$

$adaptive : coeff = \{0.15, \mathbf{0.20}, \ldots, 0.25, \ldots\}$

Hence, in LCSS metric calculation algorithm the static and adaptive parameters δ and ε were implemented. The following relations were chosen as the functional relationship between the ε parameter and the distance between the trajectory point and the camera:

$$\delta = coeff_{\delta} * \min\left(t1.length(), t2.length()\right) \tag{6.1.1}$$

$$static : \quad \varepsilon_x = coeff_{\varepsilon} * (maxX - minX) \tag{6.1.2}$$

$$static : \quad \varepsilon_y = coeff_{\varepsilon} * (maxY - minY) \tag{6.1.3}$$

$$adaptive : \quad \varepsilon_x = coeff_{\varepsilon} * \frac{(maxX - minX)}{distToCP} \tag{6.1.4}$$

$$adaptive : \quad \varepsilon_y = coeff_{\varepsilon} * \frac{(maxY - minY)}{distToCP} \tag{6.1.5}$$

where $distToCP$ is calculated as an Euclidean distance between the given point of the trajectory and the point where the camera is located at the current intersection, and $coeff_{\varepsilon}$ is the coefficient, the value of which will be selected experimentally from the set of the above mentioned. The following will show the results for the highlighted values of the $coeff_{\varepsilon}$ parameter.

In this case, when comparing the clustering results using the static parameter ε, best results were obtained with the value of coefficient $coeff = 20.0$, for the adaptive parameter – for the value of coefficient $coeff = 20.0$.

The values of the static parameter ε for each of the axes with different values of the coefficients are presented in Table 6.1.6.

Table 6.1.6: Values of static ε parameter for different coefficient values

static ε

	$\boldsymbol{\varepsilon}$ $(coeff = 0.10)$				$\boldsymbol{\varepsilon}$ $(coeff = 0.15)$			
X :								
	avg	=	123.7	(pixels)	avg	=	186	(pixels)
Y :								
	avg	=	56	(pixels)	avg	=	86	(pixels)

The statistics of the minimum, average and maximum values of the adaptive ε for each of the axes are presented in Table 6.1.7.

Table 6.1.7: Values of adaptive ε parameter for different coefficient values

adaptive ε

$\boldsymbol{\varepsilon}$ $(coeff = 20.0)$				
X :				
	max	=	351.397	(pixels)
	avg	=	187.32	(pixels)
	min	=	23.248	(pixels)
Y :				
	max	=	151.252	(pixels)
	avg	=	80.63	(pixels)
	min	=	10.004	(pixels)

As it can be seen from the given tables, static values of the ε parameter at the coefficient $coeff_{\varepsilon} = 0.15$ are approximately equal to the average values of the adaptive ε values at the coefficient $coeff_varepsilon = 20.0$. Consequently, it can be expected that the static ε values at this coefficient value will show the best results.

The comparison of static ε values was carried out with the coefficients 0.10 and 0.15. The experimental results for the data from the first and second intersections are presented in Tables 6.1.8-6.1.9. Visualization of corresponding clustering results is given in Appendix D.

According to the table, static ε gives the best results for the amount of 9 resulting clusters. The best options, which were selected, are highlighted in bold.

Data from the second intersection ($2.txt$) was used to carry out 3 experiments with a different number of final clusters: 7, 8 and 9. According to the experimental results, merging up to 8 clusters demonstrated more accurate data partitioning and higher DI values. The results of clustering on the example of data from the first and second intersections are shown in Figures 6.1.5 – 6.1.6, data clusters are displayed with different colors. On Figure 6.1.6a it is apparent that it is possible and advisable to further merge the clusters represented by yellow and green. Figure 6.1.6b depicts the result of further clustering (in this case, combining clusters), in which the two specified clusters were combined into one based on the small value of the distance between them (high degree of similarity). Figure 6.1.6c shows the result of subsequent merging of two clusters up to the final 7. It can be seen that a green cluster was merged into the red cluster, while visually that does not correspond to the actual similarity of the clusters: according to the visual test, the yellow and light pink clusters are the closest to be merged. Despite the fact that value of the DI index turned out to be higher, the visual test indicates that the obtained clusters are unsatisfactory. Thus, the most logical clustering option is clustering up to 8 clusters.

Table 6.1.8: Comparison of DI values for static ε parameter ($1.txt$)

# of clusters	$(coeff = 0.1)$ DI	comment	$(coeff = 0.15)$ DI	comment
		$1.txt$		
7	0.88	for the right line pattern (cluster) of trajectories representing straightward driving is joined with the pattern of turning to the right (light blue); pattern of horizontal driving from the left to the right is joined with the pattern of turning (red)	0.68	imprecise distant trajectories; turning to the right from the right line is not detected
8	0.83	for the right lane, the pattern (cluster) of straightward movement is merged with the pattern of turning to the right from the last right line (light pink)	**0.63**	**the most distant cluster is merged with the cluster of turning to the right (red)**
9	**0.77**	**recognizable pattern of turning to the right from the last right line; distinguishable patterns of a straightward moving from left to right and turning to the right from the last right line**	0.62	anomalous cluster in the center of the image (pink) is located on top of the cluster, representing the straightward movement from a distance

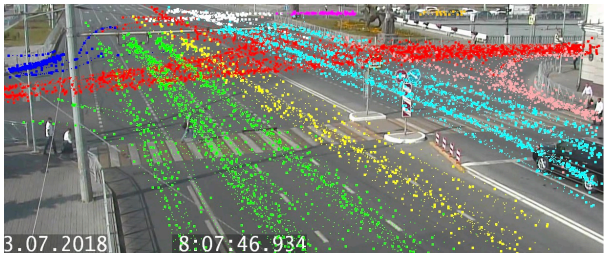
Validation of the resulting clusters was carried out using the DI metric, the following values were obtained:

- 9 clusters –> DI = 0.94,
- **8 clusters –> DI = 0.95**,
- 7 clusters –> DI = 0.97 (visually invalid clusters).

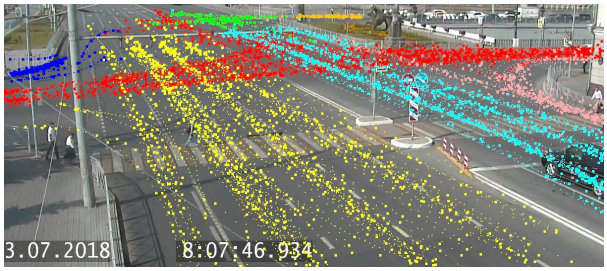
However, despite the high DI value, it can be noticed that the clustering result contains errors: blue key points corresponding to the blue cluster of trajectories are superimposed on the trajectories of the red cluster. A possible reason for such a behavior is a high degree of

Table 6.1.9: Comparison of DI values for static ε parameter ($2.txt$)

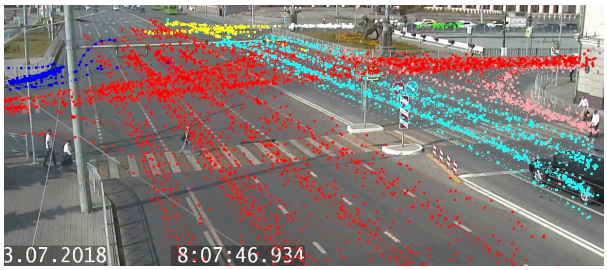
# of clusters	$(coeff = 0.1)$ DI	comment	$(coeff = 0.15)$ DI	comment
		$2.txt$		
7	0.80	central cluster comprises too distant (different) trajectories, including straightward driving and turning (red)	0.99	trajectory patterns, representing straightward movement from right to left and movement from left upward (to the left), are indistinguishable (blue)
8	0.80	major turns are distinguishable, separate cluster for long-distance trajectories	0.99	trajectory patterns of turning from right to top (to the right) and from right to bottom (to the left) are indistinguishable
9	**0.78**	**a separate cluster for a straightward movement without rounding (pink)**	**0.99**	**both aforementioned clusters are distinguishable**



(a) 9 resulting clusters, DI = 0.94



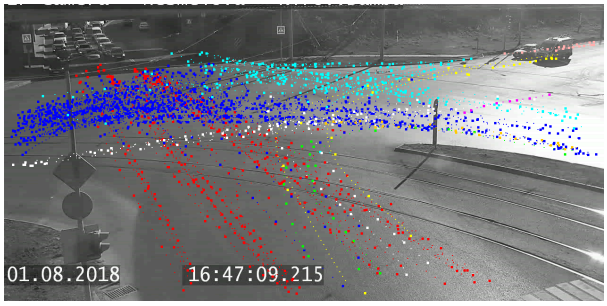
(b) 8 resulting clusters, DI = 0.95



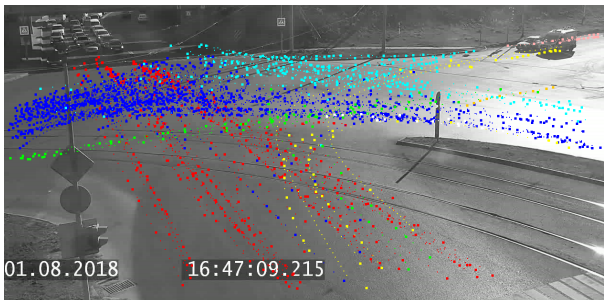
(c) 7 resulting clusters, DI = 0.97

Figure 6.1.5: Clustering results for adaptive ε, $coeff_\varepsilon = 20.0$ ($1.txt$), Polynomial Regression approximation

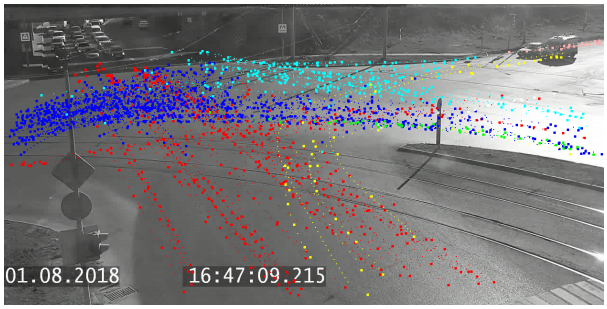
similarity between these trajectories and trajectories from the blue cluster in the upper left half of the figure: this area is represented by a dense accumulation of blue dots.



(a) 9 resulting clusters, DI = 0.94



(b) 8 resulting clusters, DI = 0.95



(c) 7 resulting clusters, DI = 0.97

Figure 6.1.6: Clustering results for adaptive ε, $coeff_{\varepsilon} = 20.0$ ($2.txt$), Polynomial Regression approximation

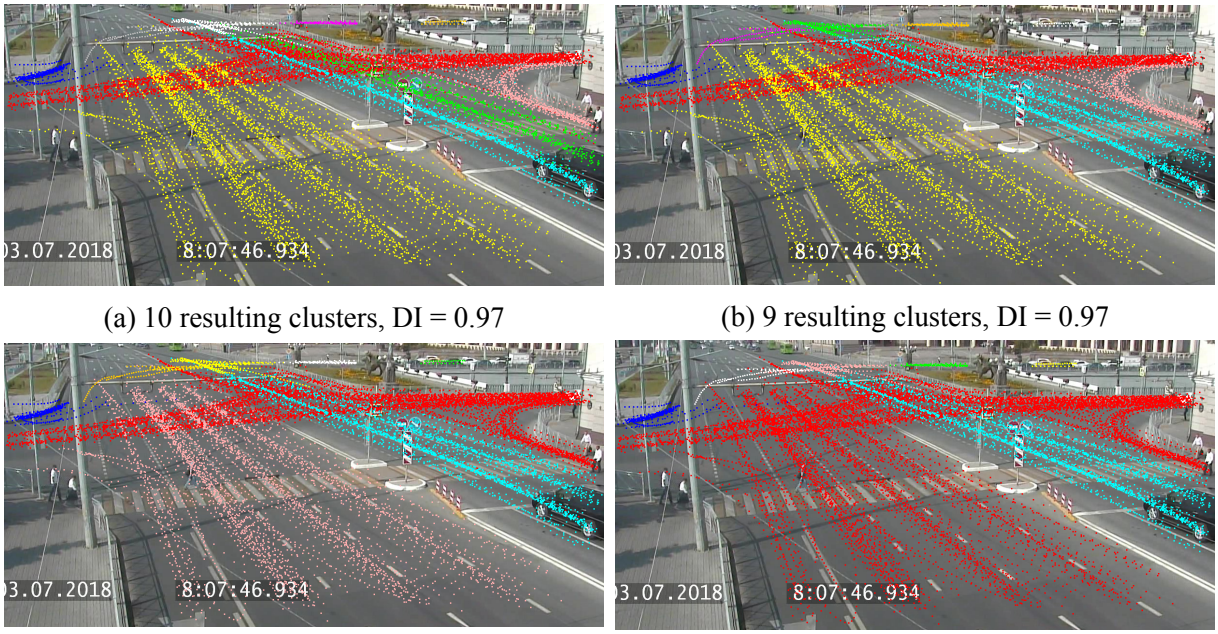
Approximation Methods Comparison

Clustering was performed and evaluated using filtered trajectories approximated by both Polynomial Regression and Douglas-Peucker N algorithms. Results of clustering using an adaptive ε with the $adapt\_coeff = 20.0$ are presented on Figure 6.1.7.

As it can be seen from the comparison of Figures 6.1.5 and 6.1.7, clustering the trajectories after Douglas-Peucker N approximation gives more precise results. While considering the images with 9 resulting clusters, in the second case algorithm specified a distinct cluster for the trajectories on the upper part of image related to vehicles with turning trajectories, while in the first case these trajectories were incorrectly included into a red cluster.

Linkage Methods Comparison

Different types of linkage methods, which are used in clustering to calculate between-clusters distance, were described in details in previous chapters. According to the tests performed by authors in [28], the single linkage method showed the best results and, in view of this, was chosen as a linkage method in the current work. However, evaluations were performed using all three linkage methods in application to used data, and comparison revealed that average or complete linkage methods demonstrate much more accurate results than single linkage method (Figures 6.1.8 – 6.1.9). Clustering with single linkage method ends up with almost all trajectories merged into one enormous cluster.



(a) 10 resulting clusters, DI = 0.97

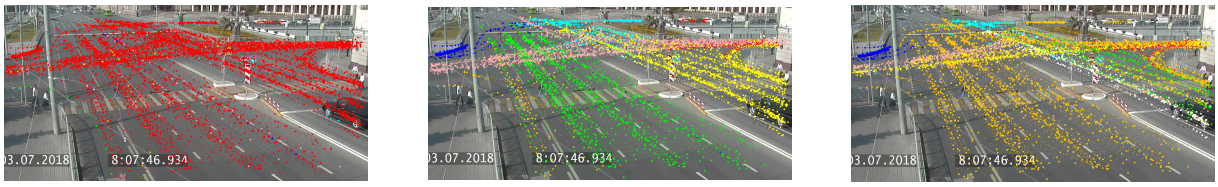
(b) 9 resulting clusters, DI = 0.97

(c) 8 resulting clusters, DI = 0.98

(d) 7 resulting clusters, DI = 0.99

Figure 6.1.7: Clustering results for adaptive ε, $coeff_\varepsilon = 20.0$ $(1.txt)$, Douglas-Peucker N approximation, $N = 8$

In this section all the results were obtained using trajectories, approximated by Polynomial regression method.

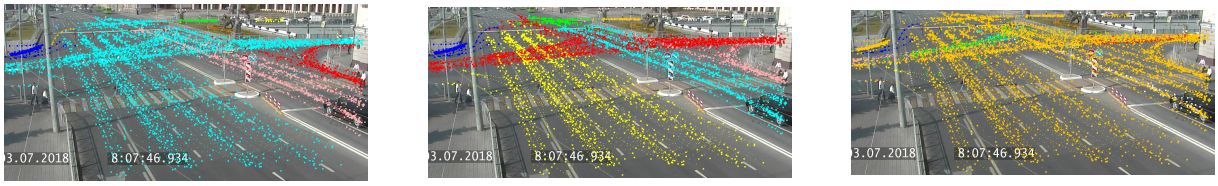


(a) single linkage method, DI = 0.14

(b) average linkage method, DI = 0.63

(c) complete linkage method, DI = 1.0

Figure 6.1.8: Comparison of clusters for different linkage methods, static $\varepsilon = 0.15$ $(1.txt)$



(a) single linkage method, DI = 0.57

(b) average linkage method, DI = 0.97

(c) complete linkage method, DI = 1.0

Figure 6.1.9: Comparison of clusters for different linkage methods, static $\varepsilon = 0.15$ $(1.txt)$

Summary

From the above it follows that based on the results of both the quantitative assessment of the DI metric and the visual test resulting clusters are more accurate when clustering till distributing data across 8 clusters for data from both intersections. In other words, 7 clusters are not enough to describe the vehicle trajectory patterns at these intersections. However, when using the static ε parameter, clustering is more accurate for 9 clusters for data from both intersections.

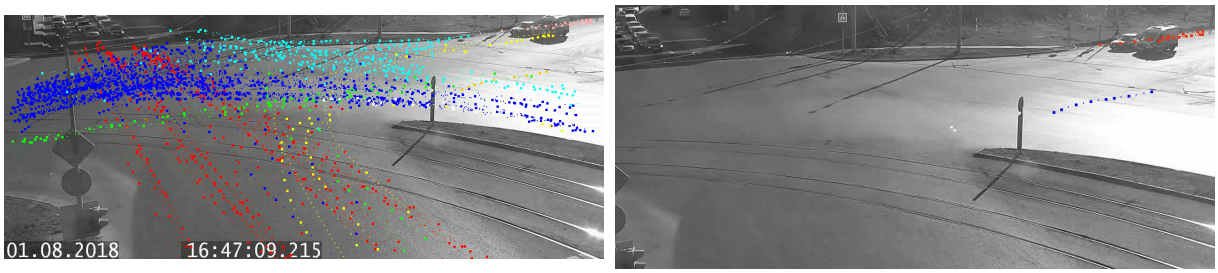
Moreover, according to the results, the use of adaptive values for the ε parameter improved the clustering results significantly.

6.1.4 Clusters Classification Results

As it was described in the previous chapter, clusters with small cardinalities are considered as clusters of anomalous trajectories. Figures 6.1.10 – 6.1.11 depict normal and anomalous clusters for data from first and second intersections respectfully.

(a) all clusters (b) anomalous clusters

Figure 6.1.10: Normal and anomalous clusters classification results ($1.txt$)



(a) all clusters (b) anomalous clusters

Figure 6.1.11: Normal and anomalous clusters classification results ($2.txt$)

6.1.5 Trajectories Classification Results

6.2 Evaluation of Performance

This section gives details on performance evaluation and provides comparative information about time for approximation, distance calculation and clustering steps for different analyzed methods.

6.2.1 Trajectories Approximation

In following paragraphs the approximation techniques will be tested using 438 preliminary filtered trajectories from the first intersection ($1.txt$) with an average length of 29 *TPs*.

Polynomial Regression

Approximation using Polynomial Regression method works fast consuming only 256 milliseconds for the whole set of input trajectories, with an average consumption of 0,6 milliseconds per trajectory. The mentioned time includes the time needed to calculate key points, the process of finding a polynomial function itself takes 73 milliseconds. The average length of approximated trajectories equals to 7,4 *TPs*.

RDP Algorithm

Approximation using an RDP algorithm takes 55 milliseconds including calculation of additional middle points. The approximation results in trajectories with lengths within 7 and 12 *TPs*, with an average of 7,96 points.

Douglas-Peucker N Algorithm

Approximation using a Douglas-Peucker N algorithm for $N = 8$ takes 223 milliseconds, which is less than a Polynomial Regression but longer than a traditional RDP algorithm. Approximated trajectories lengths' lie within 7 and 8 *TPs*, with an average value of 7,98 *TPs* before and within 5 and 8 with an average of 7,87 *TPs* after excluding the redundant points.

Summary

Taking into consideration such criterias as average and maximum approximated trajectories lengths, quality and representativity of an approximated trajectory and time consumption while comparing aforementioned approximation methods, it can be concluded that Douglas-Peucker N should be chosen as an approximation method in this case. However, Douglas-Peucker N algorithm can not detect and reflect time-dependent behavior of a trajectory whereas Polynomial Regression approximates a trajectory with a polynomial function describing a dependency between time and spatial coordinates.

6.2.2 LCSS Distances Calculation

Following Table 6.2.1 presents the time needed to perform LCSS distances calculation for filtered trajectories approximated by Polynomial regression and Douglas-Peucker N algorithms. Calculated values are in milliseconds. It can be seen than LCSS calculation works faster on trajectories approximated by Polynomial Regression, because Polynomial Regression approximation results in trajectories of shorter lengths and LCSS distance complexity and time consumption straightforwardly depends on the trajectories length.

Table 6.2.1: Evaluation of LCSS distances calculation

approximation method	time (ms), total	avg time (ms), per pair	comment
$1.txt$			
438 trajectories, 95703 trajectory pairs, $adapt\_coeff = 20.0$, average linkage method			
Polynomial Regression	1162153 ms (19,37 min)	12,4 ms	average trajectory length 7,43 *TPs*
RDP	xxx	xxx	xxx
Douglas-Peucker N, $N = 8$	2110364 ms (35,2 min)	22,05 ms	average trajectory length 7,87 *TPs*

6.2.3 Trajectories Clustering and Classification

In Table 6.2.2 the time needed to perform clustering of filtered and approximated trajectories is given. Approximation was performed using Polynomial Regression and Douglas-Peucker N techniques.

Table 6.2.2: Evaluation of trajectories clustering

approximation method	time (min), total	avg time (sec), per pair	comment
$1.txt$			
Polynomial Regression	xxx	xxx	xxx
Douglas-Peucker N, $N = 8$	xxx	xxx	xxx

7 Conclusion & Perspectives

In this work an approach for identification of trajectory anomalies in uncertain ST data was proposed and discussed. To implement the approach and further perform an evaluation, the framework was developed. The source code of the implemented framework is available on GitHub repository [51]. For solving the task of outliers detection the clustering approach was used as a basis, specifically agglomerative hierarchical clustering algorithm. To calculate the similarity and dissimilarity between trajectories and clusters, the LCSS distance was chosen. However, as it was mentioned in previous chapters, LCSS distance calculation becomes infeasible for long trajectories. For that reason the approximation of input trajectories was performed using polynomial regression and Douglas-Peucker N algorithms. According to the evaluation results, for the case of polynomial regression the best accuracy of approximation is achieved while using the 3^{th} and 4^{th}-degree polynomial functions jointly. In case of using a Douglas-Peucker N algorithm, setting a desired trajectory length to 8 trajectory points led to best relation between the complexity of inter-trajectory distance calculation and the necessity to keep the initial trajectory form. Thereby, clustering was performed on a filtered set of approximated input trajectories using key points for each of them. The accuracy of the performed clustering was evaluated using a DI index and is equal to 0.95 value, which can be considered as an acceptable result and denotes a qualitative division into clear distinguishable clusters. Hereafter for each output cluster representative trajectory was chosen as its model to simplify the further classification of a new input trajectory.

(In this work following results were achieved:)

As a result of this work following conclusions and deductions can be drawn:

- approximation of short trajectories with a non-constant speed requires higher-order polynomial functions for approximation;
- notwithstanding that LCSS distance allows trajectories to be of different lengths, it becomes extremely computationally expensive and complex for trajectories with more that 11-12 trajectory points;
- approximation using a polynomial regression works well with the trajectory data, since it is known in advance that spatial coordinates of a trajectory are functionally dependent on the time (according to principles of physics, speed, acceleration, etc.);
- approximation using a Douglas-Peucker N algorithm works faster and more accurate in terms of keeping the spatial information and representing main curves of the initial trajectory according to calculated positional errors, but Polynomial Regression is preferable for the cases, when temporal information is needed to be preserved and analyzed;
- using the adaptive parameter values significantly increases the accuracy of results according to both visual and quantitative (DI index) tests.

Further perspectives

The implemented algorithm is designed in an offline-learning manner, that means that models of normal trajectories are learned offline beforehand and are not updated with new upcoming data on an on-going basis. The future researches can include investigating an opportunity of updating normal trajectories database in order to make the framework more adaptable to actual traffic data.

Bibliography

[1] Y. Djenouri, A. Belhadi, J. C. Lin, D. Djenouri, and A. Cano. A Survey on Urban Traffic Anomalies Detection Algorithms. *IEEE Access*, 7:12192–12205, 2019.

[2] F. Mehboob, M. Abbas, R. Jiang, A. Rauf, S. A. Khan, and S. Rehman. Trajectory Based Vehicle Counting and Anomalous Event Visualization in Smart Cities. *Cluster Computing*, 21:443–452, March 2018.

[3] C. Koetsier, S. Busch, and M. Sester. Trajectory Extraction for Analysis of Unsafe Driving Behaviour. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 42(2/W13):1573–1578, June 2019.

[4] R. Ranjith, J. J. Athanesious, and V. Vaidehi. Anomaly Detection using DBSCAN Clustering Technique for Traffic Video Surveillance. In *2015 7th International Conference on Advanced Computing (ICoAC)*, pages 1–6, December 2015.

[5] S. A. Ahmed, D. P. Dogra, S. Kar, and P. P. Roy. Trajectory-Based Surveillance Analysis: A Survey. *IEEE Transactions on Circuits and Systems for Video Technology*, 29(7):1985–1997, 2019.

[6] F. Meng, G. Yuan, S. Lv, Z. Wang, and S. Xia. An overview on trajectory outlier detection. *Artificial Intelligence Review*, February 2018.

[7] G. Atluri, A. Karpatne, and V. Kumar. Spatio-Temporal Data Mining: A Survey of Problems and Methods. *ACM Computing Surveys*, 51(4), 2017.

[8] F. Tung, J. S. Zelek, and D. A. Clausi. Goal-Based Trajectory Analysis for Unusual Behaviour Detection in Intelligent Surveillance. *Image Vision Comput.*, 29(4):230–240, March 2011.

[9] Y. Li, J. Bailey, L. Kulik, and J. Pei. Mining Probabilistic Frequent Spatio-Temporal Sequential Patterns with Gap Constraints from Uncertain Databases. In *2013 IEEE 13th International Conference on Data Mining*, pages 448–457, December 2013.

[10] G. Yuan, P. Sun, J. Zhao, D. Li, and C. Wang. A Review of Moving Object Trajectory Clustering Algorithms. *Artificial Intelligence Review*, 47(1):123–144, January 2017.

[11] A. d'Acierno, A. Saggese, and M. Vento. Designing Huge Repositories of Moving Vehicles Trajectories for Efficient Extraction of Semantic Data. *IEEE Transactions on Intelligent Transportation Systems*, 16(4):2038–2049, August 2015.

[12] V. Bogorny V. C. Fontes. Discovering Semantic Spatial and Spatio-Temporal Outliers from Moving Object Trajectories. *ArXiv*, abs/1303.5132, 2013.

[13] H. Liu, X. Li, J. Li, and S. Zhang. Efficient Outlier Detection for High-Dimensional Data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(12):2451–2461, December 2018.

[14] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41(3), July 2009.

[15] F. E. Grubbs. Procedures for Detecting Outlying Observations in Samples. *Technometrics*, 11(1):1–21, February 1969.

[16] S. K. Kumaran, D. P. Dogra, and P. P. Roy. Anomaly Detection in Road Traffic Using Visual Surveillance: A Survey. *ArXiv: Computer Vision and Pattern Recognition*, January 2019.

[17] K. Malik, H. Sadawarti, and G. Kalra. Comparative analysis of outlier detection techniques. *International Journal of Computer Applications*, 97:12–21, July 2014.

[18] D. Kumar, J. Bezdek, S. Rajasegarar, C. Leckie, and M. Palaniswami. A Visual-Numeric Approach to Clustering and Anomaly Detection for Trajectory Data. *The Visual Computer*, 33(3):265–281, March 2017.

[19] S. W. T. T. Liu, H. Y. T. Ngan, M. K. Ng, and S. J. Simske. Accumulated Relative Density Outlier Detection For Large Scale Traffic Data. In *Electronic Imaging*, volume 9, pages 1–10, 2018.

[20] P. Batapati, D. Tran, W. Sheng, M. Liu, and R. Zeng. Video Analysis for Traffic Anomaly Detection using Support Vector Machines. In *Proceedings of the 11th World Congress on Intelligent Control and Automation (WCICA)*, pages 5500–5505, March 2014.

[21] C. Piciarelli, C. Micheloni, and G. L. Foresti. Trajectory-Based Anomalous Event Detection. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(11):1544–1554, December 2008.

[22] H.-L. Nguyen, Y.-K. Woon, and W. K. Ng. A Survey on Data Stream Clustering and Classification. *Knowledge and Information Systems*, 45:535–569, December 2014.

[23] M. Ester, H.-P. Kriegel, J. Sander, and X. 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, pages 226–231. AAAI Press, 1996.

[24] R. Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, May 2005.

[25] G. F. Tzortzis and A. C. Likas. The Global Kernel k-Means Algorithm for Clustering in Feature Space. *IEEE Transactions on Neural Networks*, 20(7):1181–1194, July 2009.

[26] T. Bock. DisplayR Blog. What is Hierarchical Clustering? `https://www.displayr.com/what-is-hierarchical-clustering/`. Internet Resource, Accessed: 2020-07-05.

[27] C. R. Patlolla. Understanding the Concept of Hierarchical Clustering Technique. `https://towardsdatascience.com/understanding-the-concept-of-hierarchical-clustering-technique-c6e8243758ec`, December 2018. Internet Resource, Accessed: 2020-07-05.

[28] N. B. Ghrab, E. Fendri, and M. Hammami. Abnormal Events Detection Based on Trajectory Clustering. In *2016 13th International Conference on Computer Graphics, Imaging and Visualization (CGiV)*, pages 301–306, 2016.

[29] W. Liu, Y. Zheng, S. Chawla, J. Yuan, and X. Xing. Discovering Spatio-Temporal Causal Interactions in Traffic Data Streams. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1010–1018, New York, NY, USA, 2011. Association for Computing Machinery.

[30] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering Similar Multidimensional Trajectories. In *Proceedings 18th International Conference on Data Engineering*, pages 673–684, February 2002.

[31] T. Eiter and H. Mannila. Computing Discrete Fréchet Distance *. In *Technical report CD-TR 94/64, Technische Universitat Wien*, 1994.

[32] K. Toohey. R Package Documentation. Similarity Measures. LCSS. `https://rdrr.io/cran/SimilarityMeasures/man/LCSS.html`, May 2019. Internet Resource, Accessed: 2020-06-30.

[33] K. Toohey and M. Duckham. Trajectory similarity measures. *SIGSPATIAL Special*, 7(1):43–50, May 2015.

[34] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multidimensional time-series. *The VLDB Journal*, 15:1–20, July 2006.

[35] Zhang Zhang, Kaiqi Huang, and Tieniu Tan. Comparison of similarity measures for trajectory clustering in outdoor surveillance scenes. volume 3, pages 1135–1138, January 2006.

[36] Y. A. W. Shardt. *Statistics for Chemical and Process Engineers: A Modern Approach*, chapter 3.2 Regression Models, pages 90–104. Springer International Publishing, Cham, Switzerland, 2015.

[37] B. T. Morris and M. M. Trivedi. A survey of vision-based trajectory learning and analysis for surveillance. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(8):1114–1127, August 2008.

[38] D. Dey. Dunn index and DB index – Cluster Validity Indices. `https://www.geeksforgeeks.org/dunn-index-and-db-index-cluster-validity-indices-set-1/`. Internet Resource, Accessed: 2020-07-07.

[39] J. C. Dunn. Well-separated clusters and optimal fuzzy partitions. *Journal of Cybernetics*, 4(1):95–104, 1974.

[40] M. Pathak. DataCamp. Hierarchical Clustering in R. `https://www.datacamp.com/community/tutorials/hierarchical-clustering-R`, July 2018. Internet Resource, Accessed: 2020-06-30.

[41] Z. Ansari, M.F. Azeem, W. Ahmed, and A. Babu. Quantitative evaluation of performance and validity indices for clustering the web navigational sessions. *World of Computer Science and Information Technology (WCSIT) Journal*, 1(5):217–226, 2011.

[42] B. Desgraupes. Clustering indices. 2016.

[43] M. Chen, M. Xu, and P. Franti. A fast $o(n)$ multiresolution polygonal approximation algorithm for gps trajectory simplification. *IEEE Transactions on Image Processing*, 21(5):2770–2785, 2012.

[44] I. Hadi and M. Sabah. Behavior formula extraction for object trajectory using curve fitting method. *International Journal of Computer Applications*, 104:28–37, 10 2014.

[45] A. Pant. Introduction to Linear Regression and Polynomial Regression. `https://towardsdatascience.com/introduction-to-linear-regression-and-polynomial-regression-f8adc96f31cb`, January 2019. Internet Resource, Accessed: 2020-07-15.

[46] Ramer–Douglas–Peucker Algorithm. `https://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm`. Internet Resource, Accessed: 2020-08-25.

[47] E. de Koning. Polyline Simplification. `https://www.codeproject.com/Articles/114797/Polyline-Simplification`, June 2011. Internet Resource, Accessed: 2020-09-01.

[48] E.-H. Choi and National Highway Traffic Safety Administration. Crash Factors in Intersection-Related Crashes: An On-Scene Perspective. In *NHTSA Technical Report DOT HS 811 366*, September 2010.

[49] R. Sedgewick and K. Wayne. Polynomial Implementation. `https://algs4.cs.princeton.edu/14analysis/PolynomialRegression.java.html`. Internet Resource, Accessed: 2020-07-11.

[50] Minitab Blog Editor. Regression Analysis: How Do I Interpret R-squared and Assess the Goodness-of-Fit? `https://blog.minitab.com/blog/adventures-in-statistics-2/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit`, May 2013. Internet Resource, Accessed: 2020-07-17.

[51] Framework for Identification of Trajectory Anomalies in Uncertain Spatiotemporal Data. `https://github.com/aygulmardanova/mt-anomalies`. Internet Resource, Accessed: 2020-08-02.

List of Figures

List of Tables

List of Algorithms

List of Listings

Appendix

A. Input trajectories parsing algorithm

Listing 7.1: Input trajectories parsing algorithm

```
1  package ru.griat.rcse.parsing;
2
3  import ru.griat.rcse.entity.Trajectory;
4  import ru.griat.rcse.entity.TrajectoryPoint;
5  import ru.griat.rcse.exception.TrajectoriesParserException;
6  import org.apache.commons.io.FilenameUtils;
7
8  import java.io.FileInputStream;
9  import java.io.IOException;
10 import java.io.InputStream;
11 import java.util.ArrayList;
12 import java.util.List;
13
14 /*
15 * Parser to parse input trajectories from text file
16 *
17 * stop symbols:
18 * if meet number - read until ']' or ',' or ')'
19 * [ - check for next, if [ - check for next, if [ - isX=true, if value -
       save x,
20 * */
21 public class TrajectoriesParser {
22
23   private int openingSqBracketNumber;
24
25   private boolean trajectoryStarted = false;
26   private boolean trajectoryCoordinatesStarted = false;
27   private int indexOfT, indexOfTP;
28
29   private StringBuilder x, y, t;
30
31   private List<TrajectoryPoint> trajectoryPoints;
32   private List<Trajectory> trajectories;
33
34   public TrajectoriesParser() {
35     openingSqBracketNumber = 0;
36     indexOfT = 0, indexOfTP = 0;
37
38     x = new StringBuilder(), y = new StringBuilder();
```

```
39      t = new StringBuilder();
40
41      trajectoryPoints = new ArrayList<>();
42      trajectories = new ArrayList<>();
43    }
44
45    /**
46    * Parses input 'txt'-file
47    *
48    * @param fileName  full path to the input data file with trajectories
49    * @return          list of extracted trajectories
50    */
51    public List<Trajectory> parseTxt(String fileName) throws IOException,
       TrajectoriesParserException {
52
53      InputStream reader = new FileInputStream(FilenameUtils.normalize(
       fileName));
54      int intch;
55      while ((intch = reader.read()) != -1) {
56        char nextChar = (char) intch;
57        while ((nextChar == ',' || nextChar == ' '))
58          nextChar = (char) reader.read();
59        while (nextChar == '[') {
60          increaseOpeningSqBracketsCount();
61          nextChar = (char) reader.read();
62        }
63        while (trajectoryCoordinatesStarted) {
64          if (nextChar == '(') {
65            readCoordinates(reader);
66          }
67          nextChar = (char) reader.read();
68          if (nextChar == ']') {
69            increaseClosingSqBracketsCount();
70          }
71        }
72        nextChar = (char) reader.read();
73        while ((nextChar == ',' || nextChar == ' '))
74          nextChar = (char) reader.read();
75        if (trajectoryStarted) {
76          if (nextChar == '[') {
77            increaseOpeningSqBracketsCount();
78            readTime(reader);
79          } else {
80            throw new TrajectoriesParserException("After coordinates array
       with timestamps was expected");
81          }
```

```
82          finishProcessingTrajectory();
83        }
84      }
85
86      reader.close();
87      return trajectories;
88    }
89
90    private void processBracketsCount() {
91      if (openingSqBracketNumber == 1) {
92        trajectoryStarted = false;
93        trajectoryCoordinatesStarted = false;
94      }
95      if (openingSqBracketNumber == 2) {
96        trajectoryStarted = true;
97        trajectoryCoordinatesStarted = false;
98      }
99      if (openingSqBracketNumber == 3) {
100       trajectoryCoordinatesStarted = true;
101     }
102   }
103
104   /**
105   * Reads an x and y values from file after '(' and before next ')'
106   */
107   private void readCoordinates(InputStream reader) throws IOException {
108     char nextChar = (char) reader.read();
109     while (nextChar != ',') {
110       if (nextChar >= '0' && nextChar <= '9')
111       x.append(nextChar);
112       nextChar = (char) reader.read();
113     }
114     while (nextChar != ')') {
115       if (nextChar >= '0' && nextChar <= '9')
116       y.append(nextChar);
117       nextChar = (char) reader.read();
118     }
119     processTrajectoryPoint();
120   }
121
122   /**
123   * Reads time and saves it into already initialized trajectory by
      updating trajectoryPoint at indexOfTP position in a current
      trajectory
124   */
125   private void readTime(InputStream reader) throws IOException {
```

```
126     char nextChar = (char) reader.read();
127     while (nextChar != ']') {
128       while (nextChar != ',' && nextChar != ']') {
129         t.append(nextChar);
130         nextChar = (char) reader.read();
131       }
132       if (nextChar == ']') {
133         increaseClosingSqBracketsCount();
134       }
135       trajectoryPoints.get(indexOfTP).setTime(
136         Integer.parseInt(t.toString().trim()));
137       indexOfTP++;
138       t = new StringBuilder();
139       nextChar = (char) reader.read();
140     }
141   }
142
143   private void increaseOpeningSqBracketsCount() {
144     openingSqBracketNumber++;
145     processBracketsCount();
146   }
147
148   private void increaseClosingSqBracketsCount() {
149     openingSqBracketNumber--;
150     processBracketsCount();
151   }
152
153   /**
154   * Adds parsed trajectory into an array of output trajectories and
       prepares for the next input trajectory by resetting to 0 indexes and
       buffers
155   */
156   private void finishProcessingTrajectory() {
157     trajectories.add(new Trajectory(indexOfT, trajectoryPoints));
158     trajectoryPoints = new ArrayList<>();
159     indexOfT++;
160     indexOfTP = 0;
161     trajectoryStarted = false;
162     increaseClosingSqBracketsCount();
163   }
164
165   /**
166   * Creates a new TrajectoryPoint with collected x and y
167   * Clear the buffer
168   */
169   private void processTrajectoryPoint() {
```

```
170     TrajectoryPoint point = new TrajectoryPoint(
171       Integer.parseInt(x.toString().trim()),
172       Integer.parseInt(y.toString().trim())
173     );
174     trajectoryPoints.add(point);
175
176     x = new StringBuilder(), y = new StringBuilder();
177   }
178
179 }
```

B. Polynomial Regression initiation

Listing 7.2: Polynomial Regression initiation

```
1  // initialization
2  double[] t, x, y;
3  int degree = 3;
4  double thresholdR2 = 0.97;
5  double minR2forX = 1.0, minR2forY = 1.0;
6  int minR2forXid = -1, minR2forYid = -1;
7
8  Invocation of the polynomial regression for each trajectory
9  for (int tId = 0; tId < trajectories.size(); tId++) {
10   PolynomialRegression regressionX;
11   PolynomialRegression regressionY;
12
13   Trajectory currentTr = trajectories.get(tId);
14   t = currentTr.getTrajectoryPoints().stream()
15     .mapToDouble(TrajectoryPoint::getTime).toArray();
16   x = currentTr.getTrajectoryPoints().stream()
17     .mapToDouble(TrajectoryPoint::getX).toArray();
18   y = currentTr.getTrajectoryPoints().stream()
19     .mapToDouble(TrajectoryPoint::getY).toArray();
20   regressionX = new PolynomialRegression(t, x, degree);
21   regressionY = new PolynomialRegression(t, y, degree);
22
23 //    if regression results are not satisfactory (means that degree of
       polynomial is not enough)
24 //    try to obtain an equation with a higher degree
25   if (regressionX.R2() < thresholdR2)
26     regressionX = new PolynomialRegression(t, x, degree + 1);
27   if (regressionY.R2() < thresholdR2)
28     regressionY = new PolynomialRegression(t, y, degree + 1);
29
30   currentTr.setRegressionX(regressionX);
```

```
31    currentTr.setRegressionY(regressionY);
32
33  //     calculation of minimum R^2
34    if (regressionX.R2() < minR2forX) {
35      minR2forX = regressionX.R2();
36      minR2forXid = tId;
37    }
38    if (regressionY.R2() < minR2forY) {
39      minR2forY = regressionY.R2();
40      minR2forYid = tId;
41    }
42  }
43
44  // calculation of average R^2
45  double avgR2forX = trajectories.stream()
46    .mapToDouble(tr -> tr.getRegressionX().R2())
47    .average().getAsDouble();
48  double avgR2forY = trajectories.stream()
49    .mapToDouble(tr -> tr.getRegressionY().R2())
50    .average().getAsDouble();
51
52  // print results
53  LOGGER.info("min R2 for X is for trajectory {}: {}",
54    minR2forXid, minR2forX);
55  LOGGER.info("avg R2 for X is: {}", avgR2forX);
56  LOGGER.info("min R2 for Y is for trajectory {}: {}",
57    minR2forYid, minR2forY);
58  LOGGER.info("avg R2 for Y is: {}", avgR2forY);
```

C. Agglomerative Hierarchical Clustering

Listing 7.3: Clustering implementation

```
1  import org.slf4j.Logger;
2  import org.slf4j.LoggerFactory;
3  import ru.griat.rcse.entity.Cluster;
4  import ru.griat.rcse.entity.Trajectory;
5  import ru.griat.rcse.entity.TrajectoryPoint;
6
7  import java.util.ArrayList;
8  import java.util.List;
9
10 import static java.lang.Math.*;
11
12 import static ru.griat.rcse.misc.Utils.ADAPT_COEFF;
13 import static ru.griat.rcse.misc.Utils.OUTPUT_CLUSTERS_COUNT;
```

```
14 import static ru.griat.rcse.misc.Utils.STATIC_COEFF;
15
16 public class Clustering {
17
18     private static final Logger LOGGER = LoggerFactory.getLogger(
    Clustering.class.getName());
19
20     private List<Cluster> clusters;
21
22     private Double[][] trajLCSSDistances;
23     private Double[][] clustLCSSDistances;
24     private int minX, maxX, minY, maxY;
25     private TrajectoryPoint cameraPoint;
26
27     public Clustering(List<Trajectory> trajectories) {
28         clusters = new ArrayList<>();
29         trajLCSSDistances = new Double[trajectories.size()][trajectories
    .size()];
30         clustLCSSDistances = new Double[trajectories.size()][
    trajectories.size()];
31     }
32
33     public Double[][] getTrajLCSSDistances() {
34         return trajLCSSDistances;
35     }
36
37     public void setTrajLCSSDistances(Double[][] trajLCSSDistances) {
38         this.trajLCSSDistances = trajLCSSDistances;
39         for (int i = 0; i < trajLCSSDistances.length; i++) {
40             System.arraycopy(
41                 trajLCSSDistances[i], 0,
42                 clustLCSSDistances[i], 0,
43                 trajLCSSDistances.length);
44         }
45     }
46
47     /**
48   * set borders for an input image in terms of pixels
49   * calculate the position of a camera
50   */
51     public void setBorders(int minX, int maxX, int minY, int maxY) {
52         this.minX = minX; this.maxX = maxX;
53         this.minY = minY; this.maxY = maxY;
54         this.cameraPoint = new TrajectoryPoint(
55             (int) Math.round(0.25 * maxX),
56             (int) Math.round(0.95 * maxY));
```

```
57    }
58
59
60  /**
61  * For each trajectory in @trajectories calc the adaptive εx and εy
62  * for each trajectory point
63  *
64  * @param trajectories list of trajectories to iterate and calculate ε
     values
65  */
66  private void calcEuclDistancesToCP(List<Trajectory> trajectories) {
67        trajectories.forEach(tr ->
68        tr.getKeyPoints().forEach(kp -> {
69          double cpDist = kp.distanceTo(cameraPoint);
70          kp.setCpDist(cpDist);
71          double epsilonX = ADAPT_COEFF * (maxX - minX) / cpDist;
72          kp.setEpsilonX(epsilonX);
73          double epsilonY = ADAPT_COEFF * (maxY - minY) / cpDist;
74          kp.setEpsilonY(epsilonY);
75                })
76    );
77    }
78
79    /**
80     * Single linkage
81     * LCSS similarity measure
82     *
83     * @param trajectories  database of trajectories
84     * @return        clusters of trajectories
85     */
86    public List<Cluster> cluster(List<Trajectory> trajectories) {
87        initClusters(trajectories);
88        whileCluster(OUTPUT_CLUSTERS_COUNT);
89        printClusters();
90        validateClusters();
91        classifyClusters();
92        clustersModeling();
93        return clusters;
94    }
95
96    public void classifyTrajectories(List<Trajectory> inputTrajectories)
     throws IOException {
97    double lcssMax = 0.85;
98    List<Trajectory> anomalousTrajectories = new ArrayList<>();
99    inputTrajectories.forEach(it -> {
100     final double[] minLcss = {1.0};
```

```
101       final Cluster[] closestCluster = {null};
102       calcEuclDistancesToCP(inputTrajectories);
103
104     clusters.forEach(cl -> {
105       double curLcss = calcLCSSDist(it, cl.getClusterModel());
106       if (curLcss < minLcss[0]) {
107         minLcss[0] = curLcss;
108         closestCluster[0] = cl;
109       }
110     });
111       if (closestCluster[0] == null || minLcss[0] > lcssMax) {
112         anomalousTrajectories.add(it);
113       }
114     });
115     new DisplayImage().display(inputFileName, anomalousTrajectories);
116   }
117
118
119     /**
120   * initialize clusters with each trajectory singly
121   */
122     public void initClusters(List<Trajectory> trajectories) {
123         trajectories.forEach(trajectory ->
124               clusters.add(new Cluster(trajectory.getId(), trajectory)));
125     }
126
127     /**
128      * stopPoint - desired number of clusters to stop:
129      * if null - stop when 1 cluster is left
130      * if no joins are possible, stop.
131      */
132     public void whileCluster(Integer stopPoint) {
133         if (stopPoint == null)
134             stopPoint = 1;
135         int numOfClusters = clusters.size();
136         int id1;
137         int id2;
138         double minClustDist;
139         while (numOfClusters > stopPoint) {
140             id1 = -1;
141             id2 = -1;
142             minClustDist = Double.MAX_VALUE;
143             for (int i1 = 0; i1 < clusters.size(); i1++) {
144                 for (int i2 = i1 + 1; i2 < clusters.size(); i2++) {
145                     if (i1 != i2
```

```
146                             && clustLCSSDistances[clusters.get(i1).getId
    ()][clusters.get(i2).getId()] != null
147                             && clustLCSSDistances[clusters.get(i1).getId
    ()][clusters.get(i2).getId()] < minClustDist) {
148                         minClustDist = clustLCSSDistances[clusters.get(
    i1).getId()][clusters.get(i2).getId()];
149                         id1 = i1;
150                         id2 = i2;
151                     }
152                 }
153             }
154 //              join i1 and i2 clusters, add i1 traj-es to cluster i2
155             clusters.get(id1).appendTrajectories(clusters.get(id2).
    getTrajectories());
156 //              recalculate D for i1 and i2 lines -> set i2 line all to
    NULLs
157             recalcClustersDistMatrix(id1, id2, LinkageMethod.AVERAGE);
158 
159 //              remove i2 from 'clusters'
160             clusters.remove(id2);
161 
162             numOfClusters--;
163         }
164         printClusters();
165 
166     }
167 
168     /**
169      * Calculates LCSS distance for two input trajectories
170      * Smaller the LCSS distance - the better (0.0 - equal trajectories)
171      *
172      * @param t1 first trajectory
173      * @param t2 second trajectory
174      * @return LCSS distance for t1 and t2
175      */
176     public Double calcLCSSDist(Trajectory t1, Trajectory t2) {
177         int m = t1.length();
178         int n = t2.length();
179 
180         double delta = getDelta(m, n);
181         double epsilonX = getEpsilonX(m, n);
182         double epsilonY = getEpsilonY(m, n);
183 
184         double dist = 1 - calcLCSS(t1, t2, delta, epsilonX, epsilonY) /
    min(m, n);
185         trajLCSSDistances[t1.getId()][t2.getId()] = dist;
```

```
186         clustLCSSDistances[t1.getId()][t2.getId()] = dist;
187         return dist;
188     }
189
190
191     /**
192      * Calculates LCSS for two input trajectories
193      * Bigger the LCSS - the better
194      *
195      * @param t1        first trajectory
196      * @param t2        second trajectory
197      * @param delta     δ parameter: how far we can look in time to match
    a given point from one T to a point in another T
198      * @param epsilonX ε parameter: the size of proximity in which to
    look for matches on X-coordinate
199      * @param epsilonY ε parameter: the size of proximity in which to
    look for matches on Y-coordinate
200      * @return LCSS for t1 and t2
201      */
202     private Double calcLCSS(Trajectory t1, Trajectory t2, Double delta,
    Double epsilonX, Double epsilonY) {
203         int m = t1.length();
204         int n = t2.length();
205
206         if (m == 0 || n == 0) {
207             return 0.0;
208         }
209
210 //      check last trajectory point (of each trajectory-part recursively
    )
211 //      δ and ε as thresholds for X- and Y-axes respectively
212 //      Then the abscissa difference and ordinate difference are less
    than thresholds (they are relatively close to each other), they are
    considered similar and LCSS distance is increased by 1
213         else if (abs(t1.get(m - 1).getX() - t2.get(n - 1).getX()) <
    epsilonX
214                 && abs(t1.get(m - 1).getY() - t2.get(n - 1).getY()) <
    epsilonY
215                 && abs(m - n) <= delta) {
216             return 1 + calcLCSS(head(t1), head(t2), delta, epsilonX,
    epsilonY);
217         } else {
218             return max(
219                     calcLCSS(head(t1), t2, delta, epsilonX, epsilonY),
220                     calcLCSS(t1, head(t2), delta, epsilonX, epsilonY)
221             );
```

```
222         }
223     }
224
225     /**
226      * Calculates shortened trajectory by excluding last trajectory
    point
227      *
228      * @param t trajectory
229      * @return trajectory without last trajectory point
230      */
231     private Trajectory head(Trajectory t) {
232         Trajectory tClone = t.clone();
233         tClone.getTrajectoryPoints().remove(tClone.length() - 1);
234         return tClone;
235     }
236
237     /**
238      * calc δ
239      *
240      * @param m length of first trajectory
241      * @param n length of second trajectory
242      * @return δ value
243      */
244     private Double getDelta(int m, int n) {
245         return 0.5 * min(m, n);
246     }
247
248     /**
249      * calc ε for X
250      *
251      * @param m length of first trajectory
252      * @param n length of second trajectory
253      * @return ε value
254      */
255     private Double getEpsilonX(int m, int n) {
256         return 0.1 * (maxX - minX);
257     }
258
259 private Double getEpsilonX(TrajectoryPoint tp1, TrajectoryPoint tp2,
    LinkageMethod method) {
260     switch (method) {
261       case SINGLE:
262         return Math.min(tp1.getEpsilonX(), tp2.getEpsilonX());
263       case AVERAGE:
264       return (tp1.getEpsilonX() + tp2.getEpsilonX()) / 2;
265       case MAXIMUM:
```

```
266         return Math.max(tp1.getEpsilonX(), tp2.getEpsilonX());
267       }
268       return STATIC_COEFF * (maxX - minX);
269     }
270
271       /**
272        * calc ε for Y
273        *
274        * @param m length of first trajectory
275        * @param n length of second trajectory
276        * @return ε value
277        */
278       private Double getEpsilonY(int m, int n) {
279           return 0.1 * (maxY - minY);
280       }
281
282     private Double getEpsilonY(TrajectoryPoint tp1, TrajectoryPoint tp2,
       LinkageMethod method) {
283       switch (method) {
284         case SINGLE:
285           return Math.min(tp1.getEpsilonY(), tp2.getEpsilonY());
286         case AVERAGE:
287           return (tp1.getEpsilonY() + tp2.getEpsilonY()) / 2;
288         case MAXIMUM:
289         return Math.max(tp1.getEpsilonY(), tp2.getEpsilonY());
290       }
291       return STATIC_COEFF * (maxY - minY);
292     }
293
294       /**
295        * At each step calc a distance matrix btwn clusters
296        * Merge two clusters with a min dist -> requires an update of the
       dist matrix
297        * because of the implementation: clusterId1 < clusterId2
298        *
299        * @param clusterId1 index of left joined cluster in clusters list (
       remained cluster)
300        * @param clusterId2 index of right joined cluster in clusters list
       (removed cluster)
301        * @param method   linkage method (SINGLE, AVERAGE or MAXIMUM)
302        */
303       private void recalcClustersDistMatrix(int clusterId1, int clusterId2
       , LinkageMethod method) {
304           for (int i = 0; i < clusterId1; i++) {
305               clustLCSSDistances[clusters.get(i).getId()][clusterId1] =
```

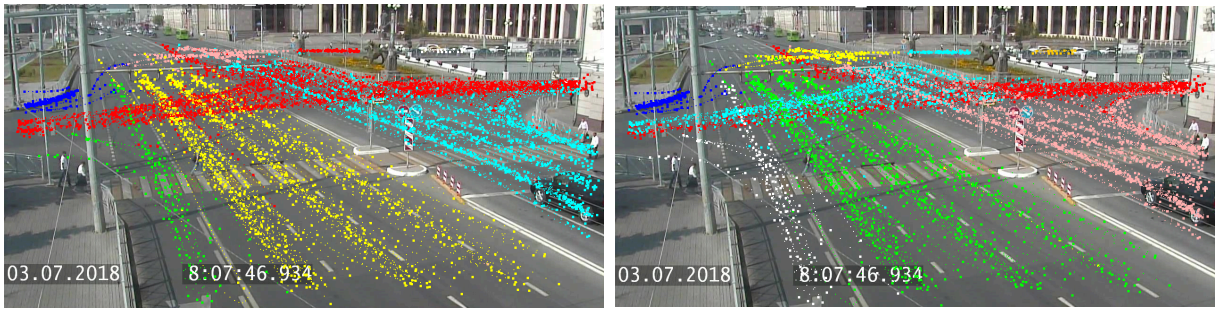
```
306             calcClustersDist(clusters.get(i), clusters.get(clusterId1)
    );
307         }
308         for (int j = clusterId2; j < clusters.size(); j++) {
309             clustLCSSDistances[clusterId2][clusters.get(j).getId()] =
310               calcClustersDist(clusters.get(clusterId2), clusters.get(j)
    , method);
311         }
312         clustLCSSDistances[clusters.get(clusterId1).getId()][clusters.
    get(clusterId2).getId()] = null;
313      }
314
315      /**
316       * Calculates inter-clusters distance for two input clusters
317       * using 'single-link' linkage method:
318       * the between-cluster distance == the min distance btwn two
    trajectories in the two clusters
319       *
320       * @param cluster1  first cluster
321       * @param cluster2  second cluster
322       * @param method  linkage method (SINGLE, AVERAGE or MAXIMUM)
323       *
324       * @return      distance between clusters
325       */
326      private Double calcClustersDist(Cluster cluster1, Cluster cluster2,
    LinkageMethod method) {
327         double dist = 0.0;
328      switch (method) {
329        case SINGLE: {
330          dist = Double.MAX_VALUE;
331          for (Trajectory trajectory1 : cluster1.getTrajectories()) {
332            for (Trajectory trajectory2 : cluster2.getTrajectories()) {
333              Double lcssDist = trajLCSSDistances[trajectory1.getId()][
    trajectory2.getId()];
334              if (lcssDist != null && lcssDist < dist)
335                dist = lcssDist;
336            }
337          }
338          break;
339        }
340        case AVERAGE: {
341          int count = 0;
342          for (Trajectory trajectory1 : cluster1.getTrajectories()) {
343            for (Trajectory trajectory2 : cluster2.getTrajectories()) {
344              Double lcssDist = trajLCSSDistances[trajectory1.getId()][
    trajectory2.getId()];
```

```
345             if (lcssDist != null) {
346               dist += lcssDist;
347               count++;
348             }
349           }
350         }
351         dist = dist / count;
352         break;
353       }
354       case MAXIMUM: {
355         dist = Double.MIN_VALUE;
356         for (Trajectory trajectory1 : cluster1.getTrajectories()) {
357           for (Trajectory trajectory2 : cluster2.getTrajectories()) {
358             Double lcssDist = trajLCSSDistances[trajectory1.getId()][
    trajectory2.getId()];
359             if (lcssDist != null && lcssDist > dist)
360               dist = lcssDist;
361           }
362         }
363         break;
364       }
365     }
366     return dist;
367   }
368
369   /**
370   * Dunn's Validity Index (DI) = dist_min / diam_max
371   * dist_min = min inter-cluster distance (minimum distance between two
    clusters;
372   * single-linkage -> min distance between two trajectories from wo
    clusters)
373   * diam_max = max intra-cluster distance (maximum distance between two
    farthermost trajectories)
374   */
375   private void validateClusters() {
376     clusters.forEach(cluster -> cluster.getTrajectories().sort(
    Comparator.comparing(Trajectory::getId)));
377
378     double minDist = Double.MAX_VALUE;
379     for (int i = 0; i < clusters.size(); i++) {
380       for (int j = i + 1; j < clusters.size(); j++) {
381         if (clustLCSSDistances[clusters.get(i).getId()][clusters.get(j).
    getId()] < minDist)
382           minDist = clustLCSSDistances[clusters.get(i).getId()][clusters.
    get(j).getId()];
383       }
```

```
384     }
385
386     double maxDiam = clusters.stream().mapToDouble(cluster -> {
387       double maxDist = 0;
388       for (int i = 0; i < cluster.getTrajectories().size(); i++) {
389         for (int j = i + 1; j < cluster.getTrajectories().size(); j++) {
390           if (trajLCSSDistances[cluster.getTrajectories().get(i).getId()
    ][cluster.getTrajectories().get(j).getId()] > maxDist)
391             maxDist = trajLCSSDistances[cluster.getTrajectories().get(i).
    getId()][cluster.getTrajectories().get(j).getId()];
392         }
393       }
394       return maxDist;
395     }).max().getAsDouble();
396
397     double DI = minDist / maxDiam;
398     LOGGER.info(String.format("DI = %.2f", DI));
399   }
400
401   private void clustersModeling() {
402     for (Cluster c: clusters) {
403       if (c.getTrajectories().size() == 1) {
404         c.setClusterModel(c.getTrajectories().get(0));
405         continue;
406       }
407       Trajectory model = null;
408       double avg = Double.MAX_VALUE;
409       for (Trajectory t: c.getTrajectories()) {
410         double sum = 0.0;
411         for (Trajectory t1: c.getTrajectories()) {
412           if (t1 != t)
413             sum += t.getId() < t1.getId() ? trajLCSSDistances[t.getId()
    ][t1.getId()] : trajLCSSDistances[t1.getId()][t.getId()];
414         }
415         double curAvg = sum / (c.getTrajectories().size() - 1);
416         if (curAvg < avg) {
417           model = t;
418           avg = curAvg;
419         }
420       }
421       c.setClusterModel(model);
422     }
423   }
424
425   private void classifyClusters() {
```

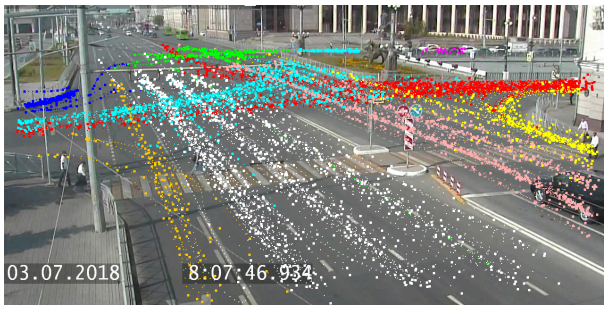
```
426     List<Integer> cardinalities = clusters.stream().map(cl ->   cl.
        getTrajectories().size()).sorted().collect(Collectors.toList());
427     double limit = Quantiles.quartiles().index(1).compute(cardinalities)
        ;
428     clusters.forEach(cl -> {
429       cl.setNormal(cl.getTrajectories().size() >= limit);
430     });
431   }
432
433 }
```

D. Visualization of clustering results for static ε



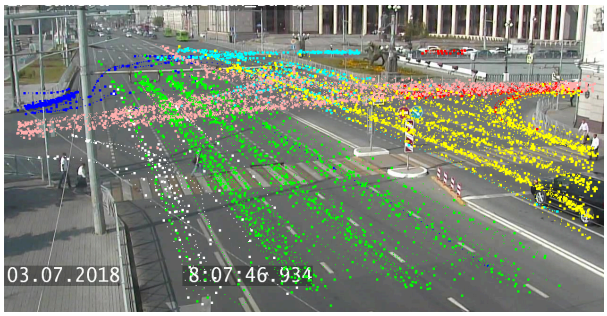
(a) 7 clusters, DI = 0.88

(b) 8 clusters, DI = 0.83

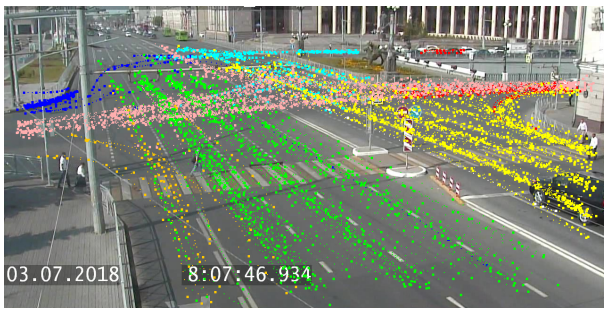


(c) 9 clusters, DI = 0.77

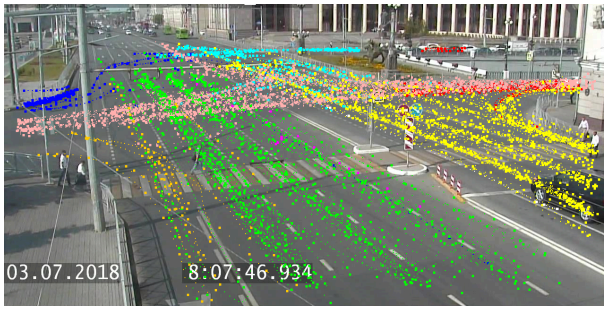
Figure 7.0.1: Clustering results for static ε, $coeff_{\varepsilon} = 0.1$ $(1.txt)$



(a) 7 clusters, DI = 0.68

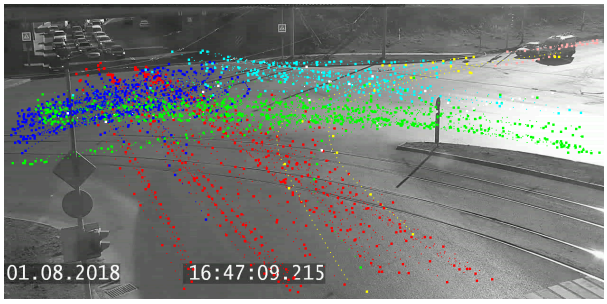


(b) 8 clusters, DI = 0.63

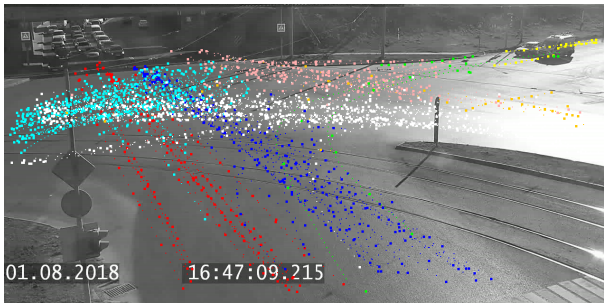


(c) 9 clusters, DI = 0.62

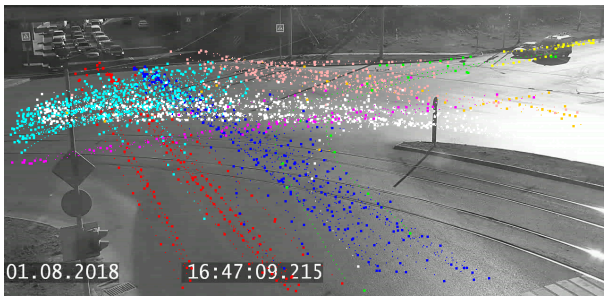
Figure 7.0.2: Clustering results for static ε, $coeff_{\varepsilon} = 0.15$ $(1.txt)$



(a) 7 clusters, DI = 0.80

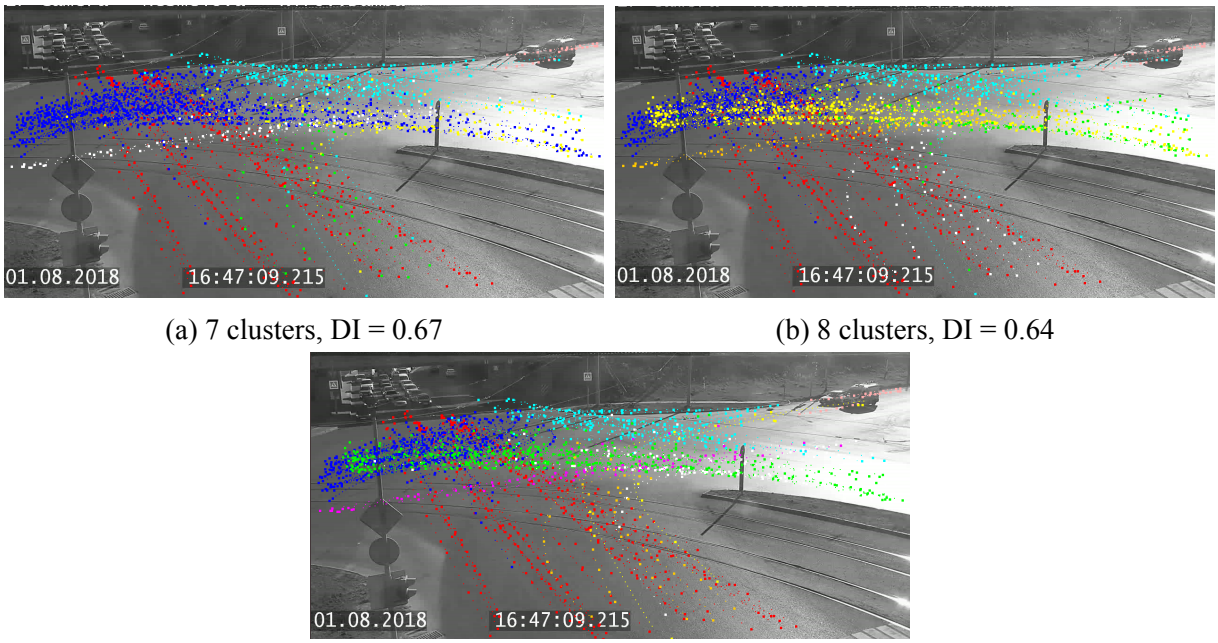


(b) 8 clusters, DI = 0.80



(c) 9 clusters, DI = 0.78

Figure 7.0.3: Clustering results for static ε, $coeff_{\varepsilon} = 0.1$ $(2.txt)$



(a) 7 clusters, DI = 0.67

(b) 8 clusters, DI = 0.64

(c) 9 clusters, DI = 0.60

Figure 7.0.4: Clustering results for static ε, $coeff_{\varepsilon} = 0.15$ $(2.txt)$