VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
MODELLING AND DATA ANALYSIS MASTER'S STUDY PROGRAMME

Master's thesis

Outlier Detection in Multidimensional Streaming Data

Išskirčių identifikavimas daugiamačiuose srauto duomenyse

Lina Ribokaitė

Supervisor PhD Jolita Bernatavičienė

Vilnius, 2021

Table of contents

Acknowledgement

The author is thankful for the HPC resources provided by the IT APC at the Faculty of Mathematics and Informatics of Vilnius University Information Technology Research Center.

Abstract

Outlier detection is an important task in many areas such as fraud detection, network analysis, sensor data analysis, etc. With increasing demand of stream data analysis, there is a need of efficient algorithm that can detect outliers in real-time. This master thesis aims to develop existing outlier detection algorithms when processing multidimensional streaming data.

The research consists of overview on present outlier detection methods and will focus on distance-based outlier detection algorithms for stream data. Four distance-based outlier detection algorithms were chosen to analyse in detail and some improvements were suggested.

First improvement is related to implementation of Support Vector Machines. By using SVM we can retrieve useful information about the behaviour of the data and use it in outlier detection. In the experimental part it is proved that the combination of distance-based algorithms and SVM improves outlier detection accuracy, mostly by increasing precision.

Another improvement is related to the output of outlier detection algorithms. Typically simple outlier list is returned. In the work we tried implementing outliers clustering which provides additional information that helps to understand the behaviour and frequency of outliers in real-time. This improvement let us highlight outliers' tendencies and this is very useful information since outlier detection in stream data is usually performed by using only most recent data and not the whole dataset.

Key words: Stream Data, Outlier Detection, Distance-based Outliers with Implemented SVM, Outliers Clustering.

Santrauka

Išskirčių identifikavimas gali būti panaudojamas įvairiose srityse, tokiose kaip apgavysčių aptikimas, ryšių analizė, jutiklinių duomenų analizė ir t.t. Vis didėjant srauto duomenų analizės poreikiams, viena iš svarbių sričių yra efektyvaus išskirčių identifikavimo algoritmo sukūrimas. Kadangi ši sritis vis dar yra vystoma, šio magistrinio darbo tikslas ir yra prisidėti prie išskirčių identifikavimo algoritmų, kurie geba efektyviai apdoroti srauto duomenis, vystymo.

Magistrinis darbas susideda iš išskirčių identifikavimo algoritmų apžvalgos, detaliau fokusuojantis į vieną sritį – atstumais grįstus išskirčių aptikimo metodus. Darbe pasirinkta detaliau analizuoti keturis gerai žinomus išskirčių aptikimo algoritmus ir pasiūlyta kelios algoritmų modifikacijos versijos.

Pirmoji modifikacija yra susijusi su SVM algoritmo panaudojimu. Naudojant SVM mes galime išgauti naudingos informacijos apie analizuojamus duomenis ir ją panaudoti kartu su atstumais grįstais išskirčių aptikimo algoritmais. Eksperimentinėje dalyje parodome, kad kombinuojant atstumais grįstus algoritmus su SVM galime pagerinti išskirčių aptikimo tikslumą.

Kita pasiūlyta modifikacija yra susijusi su algoritmų grąžinamu rezultatu. Dažniausiai algoritmai grąžina paprastą sąrašą identifikuotų išskirčių. Šiame darbe pabandėme realizuoti išskirčių klasterizavimą, kurio panaudojimas leidžia išgauti papildomą informaciją, leidžiančią suprasti bendrą identifikuojamų išskirčių vaizdą, išskirčių prigimtį bei dažnį. Ši implementacija gali parodyti išskirčių tendencijas ir tai dirbant su srauto duomenimis yra svarbi informacija, nes išskirčių aptikimas dažniausiai atliekamas analizuojant ne visą duomenų aibę, bet naudojant slenkančio lango principą.

Raktiniai žodžiai: Srautiniai duomenys, Išskirčių identifikavimas, Atstumais grįstų išskirčių identifikavimas, Išskirčių aptikimas panaudojant SVM, Išskirčių klasterizavimas.

List of abbreviations and symbols

Abstract-C - Abstracted-neighbourship-based Outlier Detection using counts;
Approx-Storm - variant of Storm algorithm, which performs effective approximations in order to reduce memory usage;
DODDS - Distance-based Outlier Detection in Data Streams;
Exact-Storm - variant of Storm algorithm, where entire window is allocated in memory and exact outlier detection answer returned;
MCOD - Micro-cluster-based Continuous Outlier Detection;
RBF - Radial Basis Function;
Storm - Stream Outlier Miner;
SVM - Support Vector Machines;

c_i - centroid of ith cluster in K-means clustering;
D - dataset;
$D(n, T)$ - time-based window;
D_n - count-based window;
$\mathcal{D}(R,k)$ - set of distance-based outliers;
$\mathcal{I}(R,k)$ - set of inliers;
k - the neighbours count threshold;
k' - the number of clusters;
l - the number of features for multidimensional data point;
m - the number of objects in a dataset;
MC - micro-cluster;
mcc_i - the center of the ith micro-cluster;
n - the number of non-expired objects;
o - data point;
$o.exps$ - expiration time of k most recent neighbour of data point o;
$o.frac\_before$ - ratio between the number of preceding neighbours which are safe inliers for o and the number of safe inliers in the window;
$o.mc$ - an identifier of a micro-cluster to which object o is assigned;
$o.pn$ - list of preceding neighbours for data point o;
$o.sn$ - number of succeeding neighbours for data point o;
$o.t$ - arrival time of data point o;
$\mathcal{P}$ - complete object set;
P - partition in K-Means clustering;
PD - the list of data points that are not in micro-clusters;
R - the distance threshold for distance-based outlier detection;
$s(i)$ - a Silhouette score for choosing k' in K-Means clustering;

S - window slide (number of data points used to define the step by which a window is slided in count-based window);
T - window size for time-based window (time period);
T_{end} - time-based window end;
T_{start} - time-based window start;
w_i - linear SVM weight assigned for ith feature;
W - window size for count-based window (number of data points in a window);
W_n - window size for time-based window counted in number of data points;
ρ - controlled fraction of safe inlier.

Introduction

Outlier detection is widely discussed topic in many areas. The process of outlier detection is closely related to clustering. Clustering aims to classify data points into groups while outlier detection aims to highlight points that do not fall into any group. Generally outlier detection intends to find observations that significantly differ from the rest of the data. Since it is such a wide topic usually it is divided into areas according to the specified problem.

Outlier detection can be used in many research areas, such as statistics, data mining, sensor networks, environmental science, distributed systems, spatiotemporal mining, etc. Also it has been studied on a large variety of data types, such as high-dimensional data, uncertain data, stream data, graph data, time series data, spatial and spatiotemporal data [1].

Originally outlier detection algorithms were built for batch mode when all the data are available at once. With increasing importance of real-time decisions they were modified for working with stream data when new data arrives continuously. In this paper we focus on outlier detection in multidimensional stream data.

Definition 1. Multidimensional data. A combination of features $o = (o_1,o_2,...,o_l)$ characterizes a particular object o. Having the number of features l and the number of objects m, objects are described as $o_i = (o_{i1},o_{i2},...,o_{il}), i = 1,...,m$ where i is the order of object in the dataset. If objects $o_i, i = 1,...,m$ are described by more than one feature, they are called multidimensional data.

Definition 2. Data stream. An infinite series of data points $...,o_{i-2},o_{i-1},o_i,...$ where data point o_i is received at time $o_i.t$ is called data stream.

Outlier detection can be used for such problems as fraud detection, network intrusion detection, spam recognition, error detection in databases, unusual sensor data analysis, sudden changes in financial market, detecting abnormal transactions or medical anomaly detection. The data for listed problems can be analysed in batch mode but all of these cases are examples of continuous processes (continuous stream of data) so being able to perform an outlier detection together with incoming stream can add value to monitoring the processes.

With increasing demand of real-time decision-making, stream data analysis is becoming more and more important. However, new challenges arise when we are working with real-time data. Some outlier detection models are developed depending on the specific data type or specific problem, so when applied on different datasets the performance of a model is not as good as expected. Also since new data arrives continuously, size of the whole dataset is increasing rapidly during the time and this is the reason why resource-constraint as well as speed of computations needs to be taken into account.

Keeping in mind these challenges many outlier detection algorithms for stream data were suggested. Since importance of real-time decisions is increasing, the topic becomes more and more important and the researchers are working on the development of these algorithms. In this project I will be focusing on developing distance-based outlier detection algorithms when working with stream data.

At the beginning in the paper work we will focus on review of existing outlier detection algorithms. Later we will move on distance-based outlier detection approaches and analyse few algorithms in detail, apply them on real-world datasets and discuss their advantages and disadvantages. Further will be introduced suggested improvements, presented corresponding experimental results and compared to the results of classical distance-based outlier detection algorithms. In advance outlier clustering is introduced which helps to retrieve additional information about outliers reported by algorithms.

Basic conclusions of the thesis compose of the analysis and improvement of distance-based outlier detection algorithms for stream data. The algorithms require to define 4 parameters, some of them need to be adapted according to the dataset so one part of the experiments is focused on the impact of choosing right parameter values. General trends on how changing parameter values affect outlier detection are highlighted in the experimental part and conclusions.

Another part of experiments consists of comparing the results of classical distance-based outlier detection algorithms and modified algorithms. One of the main observations about distance-based outlier detection was that if we treat all the features of multidimensional dataset equally in some cases it becomes hard to identify the outliers because outlierness occurs by a change of few data point attributes out of many. For this reason we have decided to take advantage of Support Vector Machines and use the information retrieved from trained SVM to learn the behaviour of labeled outliers. This modification helps us to improve outlier detection accuracy.

One more suggested implementation was outliers clustering. After experimenting on predicted outliers clustering, we saw that in some situations clustering gathers similar outliers and this may be valuable information. With time such grouping can show anomaly patterns and give suggestions about the nature and frequency of outliers.

The thesis proceeds as follows. In the next section main aims and tasks of the thesis are formulated. In Section 1 theoretical part on outlier detection in stream data is presented. Section 1.1 gives literature review on existing outlier detection algorithms. Section 1.2 introduces the concept behind a group of algorithms called distance-based outlier detection. In the section 4 distance-based algorithms are presented in detail, which are Exact-Storm, Approx-Storm, Abstract-C and MCOD. Section 1.3 consists of describing suggested improvements for distance-based outlier detection algorithms. Improvements is related to implementing SVM for better outlier detection accuracy and outliers clustering for retrieving additional information about predicted outliers. Section 1.4 is dedicated to

performance measures which later on are used for evaluating outlier detection algorithms. Section 2 is assigned for experimental part. Section 2.1 presents 4 real-world datasets that are used in experimental part. Section 2.2 presents results of described algorithms applied on real-world datasets and the influence of changing algorithm parameters on overall accuracy. Section 2.3 consists of presenting results of suggested implementations. Section 2.4 shows what additional information can be retrieved from clustering predicted outliers. Concluding the second part Section 2.5 generalizes the remarks and observations from empirical section. Finally main results and conclusions are listed and discussed in final Conclusions.

Aims and tasks

Outlier detection is an important task in such areas as fraud detection, network analysis, sensor data analysis, etc. With increasing demand of stream data analysis, there is a need of efficient algorithm that can detect outliers in real-time. This master thesis aims to develop existing outlier detection algorithms when processing multidimensional streaming data. The research consists of analysis on present outlier detection methods and suggested approach.

Tasks:

- Overview existing outlier detection algorithms for streaming data;
- Analyse 3-4 selected distance-based outlier detection algorithms;
- Identify advantages and disadvantages of selected algorithms;
- Describe performance measures that are used for comparing outlier detection algorithms;
- Develop and validate selected distance-based outlier detection algorithms;
- Apply these algorithms on several real-world datasets and describe experimental results;
- Evaluate how suggested improvements influence algorithm performance.

1. Theoretical part

In this part general knowledges and definitions on the topic of outlier detection in stream data is presented. Firstly we will discuss what are the types of outlier detection algorithms. Then we will focus on distance-based outlier detection and describe four algorithms in detail. Later on some theory behind the suggested implementations will be presented. At the end we will make a short review of the performance measures that are most common for evaluating outlier detection algorithms and later will be used in experimental part.

In general case outlier is defined as a data instance that differs significantly from other observations. They can be categorized into 3 groups [1]:

1. **Global or Point Outliers** - a data point is a global outlier if it deviates significantly from the rest of the dataset. Most outlier detection methods are dedicated to point outliers.

2. **Contextual or Conditional** - a data point is contextual outlier if it deviates significantly from the rest of the dataset with respect to a specific context. The context should be specified for a certain problem. For example, if our data is temperature then date and location may be accepted as contextual attributes.

3. **Collective** - a data subset is a collective outlier if a subset deviates significantly from the entire dataset. In this case single data point can't be defined as an outlier.

The main idea of detecting outliers is measuring how different data point is compared to the so called normal data (data that are usual or expected for a certain dataset). Let's introduce notation *outlier score*.

Definition 3. Outlier score. A score that represents the degree of the deviation of a data instance is called outlier score [2].

By using this measure data points can be ranked and according to the score outliers are identified. There are various algorithms for outlier score calculations. Some of them will be presented in detail in the following sections.

Initially many outlier detection algorithms were introduced to deal with batch data (where all of the data are available before running the algorithm). Currently more and more algorithms are adapted for streaming data. In this work we focus on working with data streams when the data comes continuously.

Since stream data are not available for processing at once, calculations must be repeated over and over again with newly arriving data. Because the flow is continuous not only outlier detection accuracy is a very important performance measure for outlier detection but also velocity and memory usage.

Outlier detection problem have already been discussed by many researchers. Following section consists of literature review on the topic of outlier detection in stream data.

1.1. Literature review

Number of studies is dedicated to outlier detection topic. Most of them focus on working with batch data. However, interest on outlier detection in streaming data has increased significantly during the last decade. Vast number of outlier detection algorithms are described in the articles [1, 2, 3, 4]. According to the base of an algorithm, outlier detection methods are categorized into these groups: distribution-based, distance-based, density-based, tree-based, clustering-based and deep learning-based. They are shortly described below.

Distribution-based

Distribution-based (or Statistical-based) method [1] is motivated on probabilistic data model. Talking more precisely it is based on statistical hypothesis testing, where null hypothesis states that data point was generated from the same distribution as the normal data (*reminder: normal data is not related to normal distribution. In this work term normal data defines the data that are usual or expected for a certain dataset*). However, it is almost impossible to model the distribution accurately for multidimensional data and this is the main reason why distribution-based outlier detection models are not used very often.

Distance-based

Distance-based method [1, 2, 5, 3, 4, 6] is based on calculating distances between the data point and its neighbours. Two main parameters k and R appear in the algorithm: data point is labeled as an outlier if less than k data instances in the input data is within distance R from this data point. For detecting outliers in data streams, distance-based algorithms are often used because of their good performance. There are quite a big number of distance-based algorithms introduced but most often you can hear about these: NETS [5], Thresh-LEAP [4], DUE [4], MCOD [6], Exact-Storm [7], Approx-Storm [7], Abstract-C [8].

Density-based

Density-based method [1, 2] is based on computing densities of separate data clusters. In a batch mode an algorithm called LOF (Local Outlier Factor) was suggested. The main idea is computing outlier score for each data point based on local density around that data instance. It indicates weather the data point is in dense region or not with respect to its neighbours. Talking about streaming data an incremental version of LOF called iLOF was presented [9]. The idea is that for every incoming data point o iLOF finds its k-nearest neighbours, computes outlier score based on its neighbours outlier score and updates past data if needed. However, iLOF method requires keeping all the previous data points of a stream so it needs high memory resources. Simply deleting old data does not solve the problem because the accuracy reduces. Upgraded version of iLOF called MiLOF (Memory Efficient Incremental Local Outlier Detection) and its extension MiLOF_F was

introduced [10]. The main difference is that instead of keeping all the stream algorithms summarize outlier score of the data that exceed the memory. This way memory usage is reduced without losing the information about the old data.

Tree-based

Tree-based method [2] process data by using tree structure. An algorithm called Streaming HS-Tree [11] was proposed for outlier detection in stream data. Such tree consists of a set of nodes which holds the number of data points within a particular subspace of the data stream. Streaming HS-Tree requires constant amount of memory and performs high detection accuracy with fast model updates. Another algorithm proposed for outlier detection is RS-Forest [2]. In stream mode trees are built by randomly selecting attributes and splitting values. When tree is constructed it is traversed and the number of instances falling into each node is computed. Later on this count is used for calculating outlier score.

Clustering-based

Clustering-based method [1, 2, 10] is based on grouping the data. Depending on an algorithm, data points are described as outliers if they form small clusters or they are far away from their cluster centroid. For example, clustering-based algorithm called SVDD (Support Vector Data Description) is mentioned in a survey of outlier detection techniques [1]. The main idea is that SVDD conducts a small sphere around the normal data and uncovered data points are defined as outliers.

Deep learning-based

Deep-learning based method [2] is a very recent approach even in batch mode. However, there are some applications made on streaming data as well. An online unsupervised deep learning approach is presented in the article [12]. The idea of suggested algorithm is that firstly data goes through feature extraction system and then it is given to a neural network. As a result, application learns behaviour of a common data and by using this information outliers are detected.

These are the main classes of outlier detection algorithms in stream data. Table 1 below shows summarized literature overview with listed algorithm parameter values that were used in experimental part, measure list used for comparing results and short description of how the results were presented.

To sum up described algorithms have their advantages and disadvantages. Distribution-based outlier detection method faces the problem that data distribution is usually unknown and it is quite difficult to accurately model the distribution for multidimensional data. Tree-based methods usually create over-complex trees so it does not generalize the data well. Clustering-based method needs initial information about formation of clusters and later on the initial information may highly affect outlier detection. Deep learning-based

Table 1: Summarizes literature review of outlier detection algorithms

Authors	Category & Algorithms	Parameters	Measures	Results
M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas and Y. Manolopoulos [6]	Distance-based: COD, ACOD, MCOD, Absract-C	$W = 200000$; $S = 1$; R ={different for datasets}; $k = 10$	CPU cost, memory required, number of distance computations, other qualitative measurements	Graphical comparison of algorithms when varying parameters W,k,R and number of outliers
F. Angiulli and F. Fassetti [7]	Distance-based: Exact-Storm, Approx-Storm	$W = 10000$; R ={different for datasets}; $k = 50$	Precision, recall, running time	Graphical comparison of algorithms when varying parameter ρ (Approx-Storm)
L. Tran, L. Fan and C. Shahab [4]	Distance-based: MCOD, Exact-Storm, Abstract-C, DUE, Thresh_LEAP, Approx-Storm	$W = 10000$ (or $W = 100000$)*; $S = 500$ (or $S = 5000$); R ={different for datasets}; $k = 50$	CPU time and peak memory, precision and recall for accuracy	Graphical comparison of algorithms when varying parameters W,S,k,R and dataset dimensionality
S. Yoon, J.G. Lee and B.S. Lee [5]	Distance-based: NETS	$W = 10000$ (or $W = 100000$); $S = 500$ (or $S = 5000$); R ={different for datasets}; $k = 50$	Peak memory, average CPU time	Graphical comparison with other models
M. Saleh, C. Leckie, J. C. Bezdek, T. Vaithianathan and X. Zhang [10]	Density-based: MiLOF, MiLOF_F	$k = 10$,20,30; $c = 50$; $I_1 = 100$,$I_2 = 10$; $b = 500$; (time-based window used)	Runtime, AUC, number of data points in memory	Graphical comparison when varying parameters c,b
Swee Chuan Tan, Kai Ming Ting and Tony Fei Liu [11]	Tree-based: Streaming HS-Trees	$W = 250$; number of HS-Trees 25 $maxDepth = 15$; $sizeLimit = 2.5$;	Runtime, AUC	Comparison of algorithm modifications presented in table
A. Tuor, S, Kaplan, B. Hutchinson, N. Nichols and S. Robinson [12]	Deep learning-based: DNN and RNN models	W: from 400 to 1000; hidden layers: from 1 to 6; hidden layer dimensions: from 20 to 500; learning rate = 0.01; batch size: from 256 to 8092	Cumulative recall	Results are given by varying hyperparameters

* depending on the dataset

models need training to operate and usually requires high processing time if, for example, large neural networks for outlier detection are built. Density-based as well as distance-based methods have been proved to be effective in detecting outliers successfully, but usually requires huge amount of computations. Also they are based on the selection of nearest neighbours and needs predefined parameters. However, distance-based as well as density-based methods are most developed areas because for a long time it showed best performance in the field and also is widely applicable with different data types and different real-world problems [13].

Distance-based outlier detection method is popular because of its scalability (if you can get good result on a small database, an algorithm that is highly scalable would work well on large set as well). Another thing is that you do not need any information about data distribution. The main requirement for distance-based outlier detection algorithms is to define a distance function which will be used to find similarity among data instances. Distance-based outlier detection methods are also quite easy to understand when you want to know why certain decision about data point outlierness was made. These are the reasons why this method was chosen for further analysis. Following sections will be focused on distance-based outlier detection methods.

1.2. Distance-based outlier detection algorithms

Distance-based method is most developed area for outlier detection when working with stream data. In short this specific problem is called DODDS (Distance-Based Outlier Detection in Data Streams).

Number of studies have been performed on DODDS [1, 2, 3, 4, 6] with overviews of already existing algorithms, some improvements or newly suggested algorithms. Further main notations that one needs for understanding DODDS are presented. All used symbols and abbreviations also can be found in the section List of abbreviations and symbols.

Definition 4. Neighbour. Given a distance threshold R ($R > 0$), a data point o is a neighbour of data point o' if the distance between o and o' is not greater than R. A data point is not considered a neighbour of itself [4].

Definition 5. Distance-based Outlier. Given a dataset D, a count threshold k ($k > 0$) and a distance threshold R ($R > 0$), a distance-based outlier in D is a data point that has less than k neighbours in D [4].

Definition 6. Inlier. A data point that has at least k neighbours within distance R is called an inlier. [4].

In general we will mark our complete object dataset as $\mathcal{P}$. This set is composed of two subsets: $\mathcal{D}(R,k)$ is a set of distance-based outliers and $\mathcal{I}(R,k)$ is a set of inliers. These sets satisfy following statements: they do not overlap ($\mathcal{D}(R,k) \cap \mathcal{I}(R,k) = \emptyset$) and they complete each other ($\mathcal{D}(R,k) \cup \mathcal{I}(R,k) = \mathcal{P}$).

In this work we focus on outlier detection in stream data where new data arrives continuously. Typically it is processed using a sliding window approach which allows reducing memory usage and processing time. Since stream is arriving continuously it is impossible to keep all the data in memory and suggested sliding window approach requires keeping only the newest data. There are two types of sliding windows when working with data streams: *count-based window* (it maintains n most recent objects) and *time-based window* (it maintains all objects that have arrived during the last T time instances). Formal definitions for count-based and time-based windows are presented below.

Definition 7. Count-based Window. Given data point o_n and a fixed window size W, the count-based window D_n is the set of W data points: $o_{n-W+1}, o_{n-W+2}, ..., o_n$ [4].

Here W is measured in number of data points in a window. After performing window slide by S we will have window consisted of data points $o_{n-W+1+S}, o_{n-W+2+S}, ..., o_{n+S}$.

Definition 8. Time-based Window. Given data point o_n and a time period T, the time-based window $D(n, T)$ is the set of W_n data points: $o_{n'}$,$o_{n'+1}, ..., o_n$ with $W_n = n - n' + 1$ and $o_{n.t} - o_{n'.t} = T$ where data point o_n is received at time $o_{n.t}$ [4].

Every time-based window can have diverse number of data points depending on how many data points arrive during time period T. After performing window slide by S we will have window defined by time interval $T_{start} + S$ and $T_{end} + S$ (if we say that previous window started at T_{start} and ended at T_{end}). An object o expires after T/S slides.

When talking about distance-based outlier detection algorithms usually we need to define these 3 steps of an algorithm [5, 4]:

1. **Expired slide processing**. Expired data points are removed from the window.
2. **New slide processing**. New data points are added to the window.
3. **Outlier reporting**. Outliers are detected from present window.

To describe how these steps are managed in different distance-based algorithms we need to introduce terms of *preceding neighbour* and *succeeding neighbour* [4]. Preceding and succeeding neighbours for data point o will be stored in $o.pn$ and $o.sn$.

Definition 9. Preceding neighbour. Data point o is a preceding neighbour of a data point o' if o is a neighbour of o' and expires before o'.

Definition 10. Succeeding neighbour. Data point o is a succeeding neighbour of a data point o' if o is a neighbour of o' and o expires in the same slide or after o'.

For distance computations any appropriate distance measure can be used, e.g. Euclidean distance, Mahalanobis distance, or some other measure of dissimilarity.

Even though Mahalanobis distance is widely applicable with multidimensional data, in experimental part Euclidean distance will be used because with big datasets it is computationally less expensive than Mahalanobis distance. Euclidean distance for multidimensional data (size m with l attributes) between pair of observations (i,j) is defined by the following formula:

$$d_E(i,j) \equiv \sqrt{\sum_{k=1}^{l}(o_{ik} - o_{jk})^2} \tag{1}$$

In the following subsections 4 distance-based outlier detection algorithms will be presented: Exact-Storm, Approx-Storm, Abstract-C and MCOD. These algorithms were chosen because in some of the articles [4, 14] authors stated that they show a very good performance especially in terms of memory and time consumption.

1.2.1. Exact-Storm

Storm (Stream Outlier Miner) is distance-based outlier detection algorithm [7]. There are two variants of the algorithm: Exact-Storm and Approx-Storm. The latter will be defined in the following section 1.2.2.

Algorithm 1 Exact-Storm

```
1: function DetectOutlier(data,currentTime,W,S)
2:     // remove expired data
3:     for (o in currentWindow) do
4:         if (o.t <= currentTime − W) then
5:             currentWindow.remove(o)
6:     // process new slide
7:     for (o′ in newData) do
8:         query = getNeighbours(o′, R) //neighbours of o′ within distance R
9:         for nei in query do
10:            if isSameSlide(o′,nei) then
11:                o′.sn = o′.sn + 1
12:            else
13:                nei.sn = nei.sn + 1
14:                if size(o′.pn) < k then
15:                    o′.pn.insert(0,nei)
16:        currentWindow.add(o′)
17:    // do outlier detection
18:    for o in currentWindow do
19:        pre = 0
20:        for pn in o.pn do
21:            if (pn > currentTime − W) then
22:                pre = pre + 1
23:        if (pre + o.sn < k) then
24:            outliers.add(o)
25:    return outliers
```

Exact-Storm is the case when entire window is allocated in the memory and the exact answer of the data stream outlier query can be computed for a window. In this concept new notation *safe inlier* needs to be introduced which will be used for algorithm definition.

Definition 11. Safe inlier. An inlier that have at least k succeeding neighbours is called safe inlier [7].

In Exact-Storm algorithm to find data point within distance R range query search is applied (range query is a common operation that retrieves all records where some value is restricted using an upper and/or lower boundaries). Additionally for each data point preceding and succeeding neighbours are stored. For data point o, $o.sn$ keeps the number of succeeding neighbours and $o.pn$ keeps the list of most recent preceding neighbours (maximum list size is equal to k).

Let us define steps of Exact-Storm algorithm:

Expired slide processing. Expired data points are removed from the window. However, they are not removed from the lists of preceding neighbours.

New slide processing. When data point o' arrives with new slide, a range query is issued for identifying neighbours in distance R. The result of the range query is used to

initialize $o'.pn$ and $o'.sn$. In addition all preceding neighbours o of o' needs to increase its succeeding neighbours count $o.sn$ by 1.

Outlier reporting. When expired slide and new slide processing are done, outliers in the current window are reported. Data point o is an outlier in a defined window, if it has less than k neighbours including succeeding neighbours $o.sn$ and non-expired preceding neighbours in $o.pn$.

An advantage of exact-Storm algorithm is that there is no need to store the list of succeeding neighbours of data point o because they expire after expires o. Disadvantages of the algorithm is that it does not take into account the situation where data point o have high number of succeeding neighbours and is safe inlier, and anyway stores k preceding neighbours. Also since expired preceding neighbours are not removed from the list, algorithm is not very optimal in memory usage.

In Algorithm 1 you can find pseudo code of Exact-Storm.

1.2.2. Approx-Storm

Approx-Storm is based on Exact-Storm algorithm [7]. The main difference is that some effective approximations are introduced which give highly accurate answers with reduced memory usage.

First approximation consists of reducing number of data points stored in each window. In particular, all data points can be partitioned in outliers and inliers. Among inliers there are safe inliers, which will be inliers in any future windows. However, we can't simply remove them from calculations because it can affect newly arrived data points in a way that they will lose their neighbours. This is why the following strategy was introduced. Let us define ρ ($0 \leq \rho \leq 1$) - controlled fraction of safe inliers. o becomes a safe inlier when it has exactly k succeeding neighbours. At this moment, if number of safe inliers exceeds ρW, then some randomly selected safe inliers are removed from a window.

Second approximation consists of reducing the space for storing neighbours for each data point. Instead of storing the list $o.pn$, we only store the ratio between the number of preceding neighbours of o which are safe inliers and the number of safe inliers in the window. For each data point this ratio is stored in $o.frac\_before$. Note that $o.frac\_before$ is not exact ratio because it is calculated after randomly removing safe inliers described in the first approximation.

Let us define steps of Approx-Storm algorithm:

Expired slide processing. Expired data points are removed from the window.

New slide processing. For each new data point o' its neighbours within distance R are found and the result is used for introducing $o'.frac\_before$ and $o'.sn$. Also for each neighbour o of o', $o.sn$ is incremented.

Outlier reporting. When expired slide and new slide processing are done, outliers in the current window are reported. Data point o is an outlier in a defined window, if

$o.frac\_before * (o.t - currentTime + W) + o.sn$ is less than k.

An advantage of Approx-Storm is that approximations reduce memory requirements. Disadvantage is that time for processing the expired data is very similar to other distance-based outlier detection algorithms.

You can find pseudo code of an algorithm Approx-Storm described in Algorithm 2.

Algorithm 2 Approx-Storm

```
1: function DetectOutlier(data,currentTime,W,S)
2:     // remove expired data
3:     for (o in currentWindow) do
4:         if (o.t <= currentTime − W) then
5:             currentWindow.remove(o)
6:     // process new slide
7:     for (o' in newData) do
8:         query = getNeighbours(o', R) //neighbours of o' within distance R
9:         count_before = 0
10:        for nei in query do
11:            count_before = count_before + 1
12:            if size(o'.pn) < k then
13:                o'.pn.add(nei)
14:            nei.sn = nei.sn + 1
15:            if nei.sn >= k then
16:                if size(safeInlierList) >= W * ρ then
17:                    rand_ind = random(0,size(safeInlierList))
18:                    safeInlierList.remove[rand_ind]
19:                    currentWindow.remove[rand_ind]
20:                if nei not in safeInlierList then
21:                    safeInlierList.add(nei)
22:        o'.frac_before = count_before/size(safeInlierList)
23:        currentWindow.add(o')
24:    // do outlier detection
25:    for o in currentWindow do
26:        pre = o.frac_before * (W − currentTime + o.t)
27:        if (pre + o.sn < k) then
28:            outliers.add(o)
29:    return outliers
```

1.2.3. Abstract-C

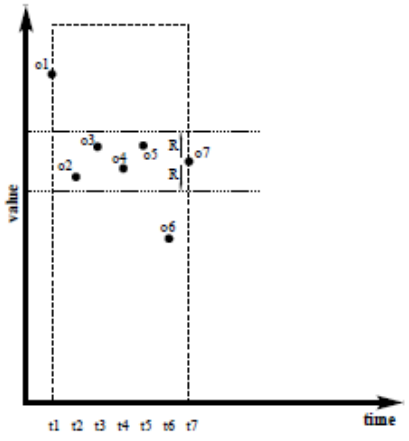
An algorithm Abstract-C (Abstracted-neighbourship-based Outlier Detection using counts), introduced in an article [8], differs from other distance-based outlier detection algorithms because in memory not the list of data point neighbours is stored but a compact summary of its neighbourships.

The challenge behind this concept was that if we store only summarized neighbourships, we lose direct access to the list of neighbours and later on we need to perform neighbours search again which increases overall computational cost. However, authors of an algorithm came up with an idea that we can predict the expiration time of any data point o_i because we know that each point participates in a constant number of windows W/S (usually for simplicity we choose parameters that W/S would be an integer).

The neighbour count described above for every data point is stored in so called *lifetime neighbour count* and is marked as $o.lt\_cnt$. Intuitively for each data point $o.lt\_cnt$ have size of W/S - the number of windows that data point o will participate in.

For better understanding of this concept let's consider an example [4] illustrated in Figure 1. Let's say window size and slide have these parameters $W = 3$ and $S = 1$. Let's talk about one data point o_3. We know that it will participate in 3 windows D_3, D_4, D_5 and will not be participating afterwards because it will be expired by that time. In D_3 (which consists of data points o_1, o_2 and o_3) o_3 has one neighbour o_2 which is within R distance from o_3. We also know that o_2 will still be a neighbour in D_4, so our lifetime neighbour count for o_3 at that moment will be $o_3.lt\_cnt = [1, 1, 0]$. In D_4, o_3 will have two neighbours because new neighbour o_4 arrives. o_4 still will be a neighbour in D_5, so at this moment lt_cnt will be updated to $o_3.lt\_cnt = [2, 1]$. In the last window D_5, o_3 has a new neighbour o_5 and o_2 is expired, so we will have $o_3.lt\_cnt = [2]$.

Figure 1: Example of DODDS when dataset D consists of 7 data points [7]



Generally distance-based outlier detection algorithm Abstract-C can be described by defining main steps of an algorithm:

Expired slide processing. Expired data points are removed from the window.

New slide processing. For each new data point o' its neighbours within R needs to be find and $o'.lt\_cnt$ is initialized. For each neighbour o that we found for o', the list of lifetime neighbour count $o.ln\_cnt$ needs to be updated.

Outlier reporting. When expired slide and new slide processing are done, outliers in the current window are reported. Data point o is an outlier in a window if it has less than

k neighbours in the current window ($o.lt\_cnt[0] < k$). Additionally for each data point o the first element of $o.lt\_cnt[0]$ is removed since we slide our window.

An advantage of Abstract-C is that it does not spend time on searching preceding neighbours in a current window for each data point. Disadvantage is that the memory requirements strongly depend on the input data stream and chosen W/S.

Below you can find pseudo code of an algorithm Abstract-C.

Algorithm 3 Abstract-C

```
1: function DetectOutlier(data,currentTime,W,S)
2:     // remove expired data
3:     for (o in currentWindow) do
4:         if (o.t <= currentTime − W) then
5:             currentWindow.remove(o)
6:     // process new slide
7:     for (o′ in newData) do
8:         query = getNeighbours(o′, R) //neighbours of o′ within distance R
9:         for nei in query do
10:            for n in W/S do
11:                if nei.t > currentTime − W and nei.t > currentTime + 1 − W +
   n ∗ S then
12:                    o′.lt_cnt[n] = o′.lt_cnt[n] + 1
13:        currentWindow.add(o′)
14:    // do outlier detection
15:    for o in currentWindow do
16:        if o.lt_cnt[0] < k then
17:            outliers.add(o)
18:        o.lt_cnt.remove[0]
19:    return outliers
```

1.2.4. MCOD

MCOD is a micro-cluster based continuous outlier detection algorithm. The idea behind the algorithm is that neighbouring information is stored in micro-clusters rather than using range queries for each data point separately. This property of MCOD adds significantly to the performance of overall outlier detection process because memory requirements are lowered when one micro-cluster keeps the neighbourhood information of many data points that fall into the same micro-cluster and it reduces the number of distance computations.

Initially R and k parameters for outlier detection are fixed. Each micro-cluster MC_i have following properties:

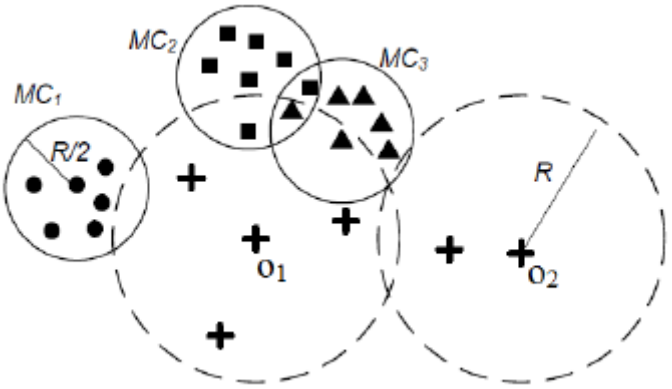
- Each micro-cluster consists of more than k data points;

- Micro-cluster is centered at one data point mcc_i and has radius of $R/2$.

You can find all additional symbols used for defining MCOD in the list of abbreviations and symbols. The radius is equal to $R/2$ because this way the distance between all data points in a micro-cluster is not bigger than R and we can say that all of these data points are inliers.

An example from an article [4] is presented in Figure 2 with parameter $k = 5$. There are 3 micro-clusters MC_1, MC_2 and MC_3. Data points that fall into micro-cluster are inliers. Some data points (represented with symbol '+') do not fall into any cluster. In the algorithm these points are stored in a separate list called PD - the list of data points which do not belong to any micro-clusters. Data points from PD can be either outliers or inliers. To find out if data point from PD list is outlier or inlier, additional search for neighbours within distance R is performed. For example, in Figure 2 neighbours of data point o_1 and o_2 are searched within circles marked with dashed line. o_1 in his dashed circle has 6 neighbours so it is an inlier. On the other hand o_2 has only 1 neighbour within distance R so it will be reported as an outlier.

Figure 2: Example of MCOD micro-clusters [4]



For each data point we have an information when the data point arrived $o.t$ and we can predict when data point expires because parameters W and S are known). Additionally if data point o belongs to a micro-cluster, this information about exact micro-cluster is stored in $o.mc$. Also for each data point we keep the expiration time of k most recent neighbours in $o.exps$ and the number of succeeding neighbours $o.sn$.

Let us define steps of an algorithm MCOD:

Expired slide processing. When new data points arrive, window slides and we have to remove outdated data points from both: micro-clusters as well as PD list. Afterwards there might be situations that some micro-cluster MC_i have less than $k + 1$ data points. If this occurs then micro-cluster MC_i is eliminated and data points that were in MC_i and still appears in active window is processed as a newly arrived data.

New slide processing. For each new data point o there are 3 possible scenarios: it might be added to the existing micro-cluster, it might become the center of a new micro-cluster or it might be added to the PD list. If o is within distance $R/2$ to the center of nearest micro-cluster then it is added to this micro-cluster. Otherwise in PD list we search for neighbours within distance $R/2$ from o. If at least k neighbours are found in PD they form a new micro-cluster with o as the cluster center. Otherwise o is added to PD list.

Outlier reporting. When expired slide and new slide processing are done, all the data points that are in PD list and have less than k neighbours within distance R are reported as outliers.

You can find pseudo code of MCOD algorithm described in Algorithm 4.

Algorithm 4 MCOD

```
1:  function DetectOutlier(data,currentTime,W,S)
2:      // remove expired data
3:      for (o in currentWindow) do
4:          if (o.t <= currentTime − W) then
5:              currentWindow.remove(o)
6:              if o.isInCluster = True then
7:                  removeFromCluster(o)
8:              else
9:                  removeFromPD(o)
10:     // process new slide
11:     for (o′ in data) do
12:         if o′ not in currentWindow then
13:             currentWindow.add(o′)
14:         nearest_center = findNearestCenter(o′)
15:         if nearest_center is not null then
16:             min_distance = distance(o′,nearest_center)
17:         if min_distance <= R/2 then
18:             //add to cluster with center nearest_center
19:             addToCluster(nearest_center, o′)
20:         else
21:             // find neighbours in PD
22:             neighboursInR2Distance = findNeighbourR2InPD(o′)
23:             if size(neighboursInR2Distance) > k then
24:                 // form new cluster with center o′
25:                 formNewCluster(o′, neighboursInR2Distance)
26:             else
27:                 addToPD(o′)
28:     // do outlier detection
29:     for o in currentWindow do
30:         if (o in PD) and (o.exps + o.sn < k) then
31:             outliers.add(o)
32:     return outliers
```

An advantage of algorithm MCOD is that pairwise distance computations between data points are more efficient. Also the memory requirements are lowered since some information for data points is kept in micro-clusters.

Four quite classical distance-based outlier detection algorithms used for stream data were presented. They are well known because of their performance. However, these algorithms have their advantages and disadvantages. In order to compare algorithms to each other Table 2 is presented.

Table 2: Advantages and disadvantages of discussed distance-based algorithms

Algorithm	Advantages	Disadvantages
Exact-Storm	It does not store list of succeeding neighbours of data point (only the count of succeeding neighbours). List of preceding neighbours is limited to store only k most recent preceding neighbours.	Is demanding in memory and CPU for storing preceding neighbour lists (also expired preceding numbers are kept). It does not take into account that data point with high number of succeeding neighbours is safe inlier and anyway stores list of preceding neighbours.
Approx-Storm	It does not store the preceding neighbours for each data point and keeps only a share of safe inliers.	Time for processing the new data does not reduce compared to other algorithms even if approximations are applied.
Abstract-C	Doesn't spend time on searching active preceding neighbours in the window for each data point.	Memory requirements strongly depend on the dataset and parameters W,S.
MCOD	Has advantage in CPU time because using micro-clusters ease distance computations. Storing the neighbourhood information in micro-clusters lowers memory usage.	MCOD loses advantage if most data points have less than k neighbours in $R/2$ and are stored in PD. It searches for all neighbours in PD for separate data point even if it highly exceeds k.

To sum up distance-based outlier detection methods for stream data is one of the most developed areas in the field. It is widely applicable with different data types and different real-world datasets. Distance-based outlier detection method is popular because of its scalability and because no information about data distribution is needed. The main requirement for distance-based outlier detection algorithms is to define a distance function which will be used for finding similarity between objects and define algorithm parameters k, R, W and S. Distance-based outlier detection methods are also quite easy to understand and explain why certain decision about data point outlierness was made.

However, distance-based outlier detection algorithms might react differently to diverse datasets (examples will be presented in the experimental part in Section 2.2). Sometimes outlierness in multidimensional data can occur in a sudden change of few data point

features. If this happens outliers might be skipped because other features behave the same way as for the majority of the data and since we are calculating the distances where each feature have equal relevance we might pass over.

To overcome this challenge we will try to implement supervised machine learning algorithm SVM that can add value to distance-based outlier detection algorithms by retrieving additional information from training subset where outliers are labeled and implementing it to the described algorithms.

1.3. Development of distance-based outlier detection algorithms

The idea behind distance-based outlier detection algorithms is calculating distances between data points. Sometimes this kind of distances are not enough to detect outliers accurately because it does not consider any initial information which can be useful for improving outlier detection.

Usually before implementing some algorithm on the dataset we have some known information (training sample) from which additional information can be retrieved. For example, if we take a case of outlier detection, beforehand we can indicate what kind of feature changes define labeled outliers in our training subset.

To retrieve additional information we will implement Support Vector Machines algorithm. We will use training subset with condition that outliers in this subset are labeled and later on experimentally check how these implementations can improve outlier detection accuracy.

Another area of development focuses on reported outliers. Traditionally outlier detection algorithms give a list of data points which are predicted to be outliers. What we are going to do is group reported outliers and investigate how frequent they arrive and what kind of common characteristics they have.

1.3.1. Implementing SVM

Support Vector Machines (SVM) is a supervised machine learning algorithm which can be used for classification or regression problems. In our case we will be talking about binary classification problem.

Definition 12. Classification. A data mining technique which is used to predict group membership for data instances is called classification [15].

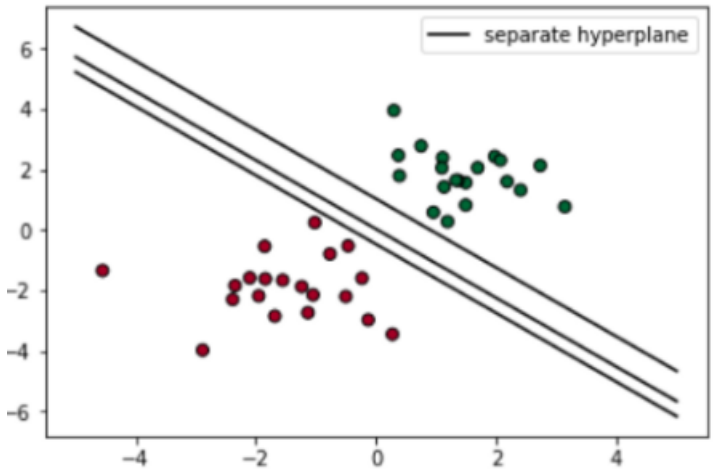
Generally SVM is a process of extracting patterns from the data. Given a training subset $\mathbf{x_i}, i = 1,...,m$ where $\mathbf{x_i}$ is characterized by l features $\mathbf{x_i} = (x_{i1},x_{i2},...,x_{il})$ and each data point have label that indicates to which of two classes $y_i \in \{-1,1\}$ (called negative and positive classes) data point belongs to, a binary SVM is trained to classify new samples into one of these classes.

The main idea of the algorithm is to find the optimal separating hyperplane (decision boundary) with the maximal margin between two classes [16] (margin in this concept is the distances between the hyperplane and the closest data points from each group to the hyperplane). In other words SVM method seeks to maximize the smallest distances of all observations to the separating hyperplane. If we working with l-dimensional space then the hyperplane is a $(l-1)$-dimensional subspace.

For better understanding let's consider 2-dimensional example in Figure 3. In this case the hyperplane is a line which is separating 2 classes (red and green data points).

Possible hyperplanes in this case are black lines. However we need to find the hyperplane which represents the separation between groups the best.

Figure 3: SVM: 2-dimensional example



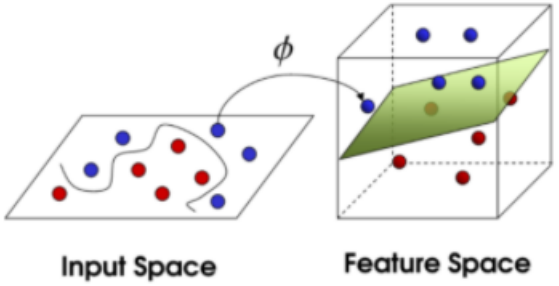
If we consider a case of linear SVM where each data point $\mathbf{x}$ is represented by a feature vector $\mathbf{x} = x_1,...,x_l$. Then the equation of the hyperplane in l-dimensional case can be defined as:

$$\mathbf{w}^T\mathbf{x} + b = \sum_{j=1}^{l} w_j x_j + b = 0, \tag{2}$$

where $\mathbf{w}$ is l-dimensional vector of coefficients corresponding to different data point features and b is a scalar.

Classification of a dataset with SVM can be linear or non-linear. When we talk about non-linear separation between groups additional function $\phi(\mathbf{x})$ needs to be introduced. If we have l-dimensional data point $\mathbf{x} \in R^l$ where R^l is a vector space with l dimensions, usually for optimisation SVM uses another more complex space called *feature space* which helps to separate the groups and find a hyperplane equation (example shown in Figure 4 where non-linear SVM needs to be applied to accurately separate classes).

Figure 4: SVM: Feature space



Without going deep into explanation we just need to know that this function maps

input features to a complex feature space $\phi(\mathbf{x}) : R^l \rightarrow R^d$ (which usually is higher dimension space than input space).

In linear SVM we will have $\phi(\mathbf{x}) = \mathbf{x}$. When different classes are not separable linearly non-linear functions are used. One of the most popular non-linear function used for training SVM is RBF (Radial Basis Function) which is defined as $\phi(\mathbf{x},\mathbf{x}') = exp(-\gamma||\mathbf{x} - \mathbf{x}'||^2)$ where γ defines the influence of new features to decision boundary and $||\mathbf{x} - \mathbf{x}'||$ is Euclidean distance between two points $\mathbf{x}$ and $\mathbf{x}'$. In the experimental part we will use linear SVM as well as non-linear (trained on RBF function).

Generally if we have a set of m multidimensional data points $\mathbf{x_i}, i = 1,...,m$, SVM intends to maximize the smallest distances of all observations to the separating hyperplane by solving following unconstrained optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m} L(\mathbf{w},b,\mathbf{x_i},y_i), \tag{3}$$

where $C > 0$ is a penalty parameter and $L(\mathbf{w},b,\mathbf{x_i},y_i)$ is a loss function.

There are 2 most common loss functions which are called L1-loss SVM and L2-loss SVM respectively:

$$\max(1 - y_i(\mathbf{w}^T\phi(\mathbf{x_i}) + b),0) \text{ and } \max(1 - y_i(\mathbf{w}^T\phi(\mathbf{x_i}) + b),0)^2. \tag{4}$$

By using the predictor for any instance we can get to which class: negative or positive, the data point $\mathbf{x_j}$ is assigned according to trained SVM. For $\mathbf{x}_j$ the decision function (predictor) is defined by a formula:

$$f(\mathbf{x}_j) = sgn(\mathbf{w}^T\phi(\mathbf{x_j}) + b). \tag{5}$$

One of the properties of linear SVM described in an article 'Feature Ranking Using Linear SVM' [16] is that coefficients $\mathbf{w} \in R^l$ obtained from (3) can be used to decide the relevance of each feature. If coefficients $|w_i|$ is large then the ith feature plays more important role in the decision function (5). Using this statement we can decide which features are most important when linear SVM is deciding (predicting) to which class data point needs to be assigned.

Once we understand the concept of Support Vector Machines we can move forward and talk about how SVM can be implemented into distance-based outlier detection algorithms.

Applying SVM for learning feature relevance

As discussed previously, if we train linear SVM on classifying a specific subset to 2 groups (outliers and not outliers), the absolute value of coefficients $\mathbf{w}$ of trained SVM can describe the relevance of the attributes for classification decision. The higher the absolute

value of coefficient, the greater importance of the feature in separating the data points with different labels. What we are going to do is using $|\mathbf{w}_i|, i = 1,...,l$ for defining weighted Euclidean distance and later on use it in the DODDS algorithms.

In further definition we consider that $w_i \geq 0, i = 1,...,l$ is the weight of ith feature. Weighted Euclidean distance between pair of observations $(\mathbf{x}_i,\mathbf{x}_j)$ with known attribute coefficients w_i is defined by the following formula:

$$d_{wE}(\mathbf{x}_i,\mathbf{x}_j) = \sqrt{\sum_{k=1}^{l} w_k(x_{ik} - x_{jk})^2} \tag{6}$$

Applying SVM for verifying outliers

Another improvement is related to double checking data point outlierness. Distance-based outlier detection methods with specific datasets may suffer from situations where false outliers are reported which is called low *precision* (the detailed concept of precision and recall will be presented in Section 1.4 Performance measures). Basically low precision says that many data points which are not outliers are reported as outliers.

To overcome this problem we will use additional filtering of outliers which are detected by using distance-based approaches. This process should reduce the number of outliers reported and should increase precision. Additional outliers filtering will be repeated in each window and can be defined in 2 steps:

1. Distance-based outlier detection algorithm reports the list of predicted outliers.
2. Reported outliers are checked with trained SVM. If SVM assigns predicted outlier to common data class (not outliers) then it is eliminated from the list of outliers.

Regarding described improvements related to SVM implementation into distance-based outlier detection algorithms, in the experimental part we will consider 3 modifications:

- Using trained linear SVM coefficients for introducing weighted distance which will be used in distance-based outlier detection algorithms;
- Using trained linear SVM for filtering outliers which are reported by distance-based outlier detection algorithms;
- Using trained non-linear SVM for filtering outliers which are reported by distance-based outlier detection algorithms (in this case to find non-linear decision boundary for training SVM we use RBF function).

In the experimental part these modifications will be implemented into distance-based outlier detection algorithms and their results will be compared. To separate modifications we will mark them with * respectively, e.g. MCOD algorithm which will use linear SVM weights for weighted distance will be marked as MCOD*, algorithm where linear SVM will be used as additional filtering will be marked as MCOD** and algorithm where non-linear SVM will be used as additional filtering will be marked as MCOD***.

1.3.2. Implementing K-Means clustering

When we are talking about outlier detection in stream data using sliding window outliers are usually identified in local concept. It means that some data points locally might be outliers but if we look in a wider concept it might be values that are rare but far between occurs within a dataset. These kind of data points are called *anomalies*.

Definition 13. Anomaly. The rare observation which differs significantly from the common behaviour (behaviour of the majority of the data) is called anomaly.

Most of the outlier detection algorithms focus on detecting outliers but do not investigate them. In some situations it might happen that outliers repeat and have some frequency which is not easily noticeable especially if we use sliding window approach. I will call this kind of outliers *anomaly patterns*. By investigating anomaly patterns we can discover that some outliers repeat and this information can stimulate some useful observations.

Increase of specific outliers may indicate that something unusual is happening, e.g. breakdown of equipment or occurrence of a particular event. This topic discussed in an article [17] simulated idea about how anomaly patterns can be discovered using outliers clustering in stream data.

The main idea is that if we cluster outliers we can get information about the nature of the specific group of detected outliers. For example, if a very dense cluster is formed, we can assume that this is some kind of anomaly pattern and we can investigate what kind of changes causes these events.

Clustering is a tool for finding similar data points within a dataset. It can be widely applicable in various areas since clustering requires only one predefined parameter - number of clusters k'. Generally clustering is the task of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups.

K-Means clustering is a simple and fast clustering technique. The procedure follows a simple and easy way to classify a given dataset through a certain number of clusters k' [18].

The main idea of the algorithm is finding k' centroids which will define separate clusters. Once centroids are chosen we need to assign each data point to specific cluster by finding which centroid is closest to a data point. This process is repeated many times

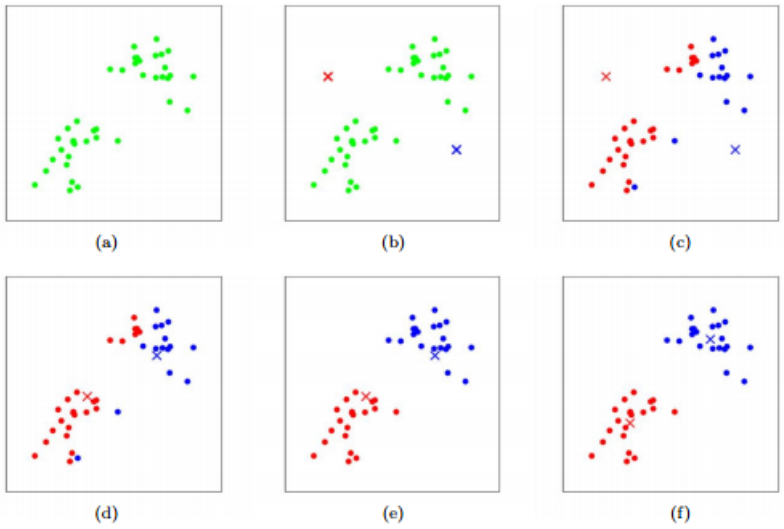
in order to adjust centroids that represent the best clustering.

Speaking more specifically the algorithm can be described by 4 steps:

1. Place k' points in the space taken from the objects that are being clustered. These points are initial centroids.
2. Assign each object to the group that has the closest centroid.
3. When all objects have been assigned recalculate the positions of the k' centroids.
4. Repeat Step 2 and 3 until the centroids no longer move.

Usually initial centroids are chosen randomly with a condition that they should not be close to each other. With number of iterations they are adjusted to form best representative clusters. Described process with $k' = 2$ is illustrated in Figure 5. Let's shortly discuss how it works. (a) illustrates initial dataset, (b) initial centroids are introduced, (c) data points assigned to a closest cluster, (d) positions of centroids adjusted, (e) data points reassigned to a closest cluster, (f) positions of centroids adjusted and so on.

Figure 5: K-Means clustering (k'=2)



The main task of the algorithm is to minimize a squared error function (distances between data points and their cluster centroids). For a partition P which consists of k' non-empty and not overlapping clusters $P_K, K = 1,...,k'$ with centroids $c_K, K = 1,...,k'$ the squared error function is defined as follows:

$$W(P,C) = \sum_{K=1}^{k'} \sum_{i \in P_K} ||\mathbf{x_i} - \mathbf{c_K}||^2, \tag{7}$$

where $\mathbf{x_i}, i = 1,...,m$ and $\mathbf{c}_K, K = 1,...,k'$ are sets of l-dimensional datapoints.

Most of the clustering algorithms are designed to investigate the grouping of data objects according to a known number of clusters k'. Identifying the number of clusters k' is an important task for any clustering problem. There are number of different proposed approaches which helps choosing best k'. We will use Silhouette Method.

The silhouette value measures how similar a point is to its own cluster compared to other clusters. The silhouette score reaches its global maximum at the optimal k', it means we need to find to which k' silhouette score has highest value.

The silhouette score $s(i)$ for each data point i is defined by a formula:

$$s(i) = \begin{cases} \frac{b(i)-a(i)}{max\{a(i),b(i)\}}, \text{ if } |C_i| > 1 \\ 0, \text{ if } |C_i| = 1 \end{cases}, \tag{8}$$

where $|C_i|$ is the number of data points in the ith data point cluster.

$a(i)$ here measures similarity of the point i to the data point j of same cluster. It is calculated by a formula:

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i,j), \tag{9}$$

where function $d(i,j)$ measures distance between two data points.

$b(i)$ is the measure of dissimilarity of point i and the point j in other cluster:

$$b(i) = \min_{i \neq j} \frac{1}{|C_j|} \sum_{j \in C_j} d(i,j). \tag{10}$$

K-Means clustering and Silhouette Method can be implemented into described outlier detection methods. Let's discuss what value can these implementations add to the outlier detection.

Applying K-Means to outliers clustering

Clustering will be implemented into distance-based outlier detection algorithms described in Sections 1.2. We will use K-Means algorithm which is rather simple and fast clustering technique. For choosing number of clusters which separates outliers groups best we will use Silhouette method. Since the patterns of detected outliers might change during the time, K-Means clustering will be trained every n_{cl} iterations (windows).

Together with clustered outliers the algorithms will present additional information about clusters:

- Number of data points belonging to a cluster.
- Average distance between cluster data points and centroid.
- Mean feature values of data points in a cluster.

After retrieving this additional information some important conclusions about the group of reported outliers can be done. We will discuss it more precisely in experimental part.

To sum up, Section 1.3 introduced improvements that could be applied on distance-based outlier detection algorithms for stream data. One of the improvements is suggested based on implementing SVM which helps retrieving additional information about labeled outliers from training subset. It is expected that these improvements will increase outlier detection accuracy. Another improvement is suggested to extend the output of outlier detection algorithms. It is suggested to implement outliers clustering which provides additional information about the similarity of outliers.

Further the possible workflow is presented which generalize the suggested improvements for distance-based outlier detection algorithms when working with stream data:

1. Train SVM on given training sample where outliers are labeled.
2. Perform modified distance-based outlier detection with one of the listed modifications (a-c):
   (a) Use linear SVM coefficients for defining weighted distance which then is used in distance-based outlier detection.
   (b) Run distance-based outlier detection and apply additional filtering for identified outliers by using linear SVM.
   (c) Run distance-based outlier detection and apply additional filtering for identified outliers by using non-linear SVM.
3. Perform K-Means clustering for predicted outliers.

In experimental part of the thesis we will use this approach with suggested additions and compare the results.

In order to compare described distance-based outlier detection algorithms and investigate how suggested implementations add value to the classical methods, performance measures needs to be discussed.

1.4. Performance measures

As described in previous sections, there are various algorithms for outlier detection in stream data. To evaluate which algorithm shows better performance we need to define measurements that allow us to compare outlier detection algorithms to each other.

Usually when talking about outlier detection in stream data we pay attention to these 3 characteristics: how well algorithm identifies outliers, how fast works an algorithm and how much memory is needed for overall outlier detection process. Further we will define what kind of indicators can be used for measuring these characteristics.

Accuracy. Accuracy defines how well an algorithm detects outliers. When dealing with outlier detection problem usually two indicators are defined for accuracy measurement: *Precision* and *Recall*. To define these measures we need to introduce context of *Confusion matrix*.

In a binary decision problem we have two different labels, we call them positive and negative. The decision made by an algorithm can be represented in a confusion matrix which is shown in Figure 6. The main idea is that confusion matrix has four categories. True positives (TP) are objects correctly labeled as positives. False positives (FP) refer to negative objects incorrectly labeled as positive. True negatives (TN) correspond to negatives correctly labeled as negative. False negatives (FN) refer to positive objects incorrectly labeled as negative. In our case positive class will be outliers and negative class inliers.

Figure 6: Confusion matrix [19]

	actual positive	actual negative
predicted positive	TP	FP
predicted negative	FN	TN

Recall measures how well algorithm identifies real outliers. It is calculated by using formula:

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

Precision measures how well algorithm identifies outliers with respect to all predicted outliers. It is calculated by using formula:

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

CPU time. When outlier detection is performed with stream data it is critical to reduce algorithm latency. CPU time is used to compare how fast algorithm processes defined data steam. CPU time defines the time consumed for running the algorithm.

Memory usage. Memory consumption is another important indicator when working with stream data. We can't store all the dataset because then we will need enormous requirements for memory storage. The smaller algorithm memory consumption the better. Maximum memory consumed (also called peak memory) is used for comparing different outlier detection algorithms. It defines the maximum memory consumption when processing stream data for outlier detection.

To generalize, Section 1 was dedicated to theoretical part behind the algorithms of outlier detection in stream data. In Section 1.1 we presented an overview on outlier detection techniques and talked what are the advantages and disadvantages of different outlier detection algorithms. Then in Section 1.2 we discussed 4 distance-based outlier detection algorithms for stream data. These algorithms will be used in experimental part. In Section 1.3 we presented necessary theory and the improvements for distance-based algorithms. In Section 1.4 performance measures were introduced. They will be used in experimental part to compare the results between algorithms.

Further in 2nd part of the paper we will apply these algorithms on chosen datasets and discuss the results.

2. Experimental results

In Section 1 theoretical part behind distance-based outlier detection algorithms for stream data were presented. Further we will move on the experimental part and describe the results observed after applying discussed algorithms on several datasets.

Our experimental part will consist of 4 main subsections which will cover description of the datasets used, results of applying selected distance-based outlier detection algorithms on the chosen datasets, discussing results of modified algorithms and comparing it to the classical distance-based outlier detection algorithms and lastly presenting the results of outliers clustering.

Let's shortly discuss what kind of experiments are presented in the experimental part.

In Section 2.2 we present the application of classical (unchanged) distance-based outlier detection algorithms for stream data. Since choice of algorithm parameters is very important we discuss how changing the parameters (increasing or decreasing parameter values) affects the results. Based on these experiments default parameter values for each dataset is defined.

Section 2.3 defines how suggested improvements for distance-based outlier detection algorithms influence the results. Firstly we introduce what information is learnt from the training subsets of each dataset. Then we discuss the results of implementing suggested approaches while comparing modified algorithms with classical (unchanged) algorithms.

In Section 2.4 we talk about outliers clustering and how it might be implemented into outlier detection algorithms when working with stream data. We learnt that in some cases (depending on the dataset) outliers clustering might give useful information about the groups of outliers detected. This information later on can be used for automatically defining detected outliers in real time.

First of all let's describe the datasets that are used in experimental part.

2.1. Datasets

For outlier detection problem it is difficult to find real-world datasets containing labeled outliers. This is why when working with development of outlier detection algorithms usually open-source datasets are used. Most of them are created from classification datasets by reducing or eliminating classes so that some of the data would differ from the majority of the data.

In this work four real-world datasets were chosen to be used. All of them are available in the UCI KDD Archive [20] as well as in Outlier Detection DataSets (ODDS) library [21] where data are specifically prepared for outlier detection problem. Different datasets in respect to the size, number of attributes and outlier rate (Table 3) were chosen to investigate and analyse how outlier detection algorithms perform using different datasets.

Table 3: Description of datasets

Dataset	Size	Attributes	Outlier rate
Shuttle	49097	9	7.00%
KDD Cup 1999	567479*	3	0.40%
Pendigits	6870	16	2.27%
Forest Cover	286048*	10	0.90%

* for these datasets a subset of 50000 data points are used in experimental part

Shuttle dataset originally is multi-class classification dataset. In our case the biggest class is defined as inliers, one class is removed and the rest data points are labeled as outliers. Modified dataset contains of 49097 records with 7% outlier rate and 9 attributes.

KDD Cup 1999 dataset consists of network data with a task of network intrusion detection. Originally dataset have 42 attributes but for outlier detection it is reduced to 3 which according to the data providers are defined as the most basic attributes (duration of connection, number of failed logs in attempts and number of connections to the same host as the current connection). It contains of 567479 records with 0.4% outlier rate. Each connection is labeled as either usual, or as an attack. In experimental part we will use a subset of 50000 data points.

Pendigits is a multiclass classification dataset having 16 attributes and 10 classes (original problem is classification of numbers from 0 to 9). The digit database is created by collecting 250 samples from 44 writers. For outlier detection problem dataset is modified in a way that one class is reduced by 90% so it would form outliers and the rest classes are accepted as inliers. Modified dataset contains of 6870 records with 2.27% outlier rate.

Forest Cover dataset is used for predicting forest cover type from cartographic variables. It takes forestry data from four wilderness areas in Roosevelt National Forest in northern Colorado. The observations are taken from 30m by 30m patches of forest that are classified as one of seven cover types. The dataset consists of several cover type classes

but we will use only two of them: one is considered as inliers and other as outliers. Initial dataset have 54 attributes but in our case we will use 10 (binary attributes are removed). Modified dataset contains of 286048 records with 0.9% outlier rate. Instances from class 2 are considered as inliers and instances from class 4 as outliers. In experimental part we will use a subset of 50000 data points.

In the Appendix 1 descriptive analysis and graphs of used datasets are placed. For each dataset two tables are presented: one describes labeled outliers and another describes the rest of the data. Also parallel coordinates graphs are presented where vertical lines represent different attributes and data points are represented as a polylines with vertices on the parallel vertical axes. Red lines in these graphs represent data points which are labeled as outliers.

Since our algorithms are calculating distances between data points, it is very important having standardized ranges for data point attributes because values may be from different range for each attribute. For simplicity scaling on the whole dataset was applied before running the algorithms.

In the following sections we will discuss the results of distance-based outlier detection algorithms applied on described datasets.

2.2. Application on real-world datasets

In theoretical part the concept of distance-based outlier detection was introduced. Four algorithms: Exact-Storm, Approx-Storm, Abstract-C and MCOD were presented in detail. In this section we will discuss results observed after these algorithms were applied on chosen datasets.

Distance-based outlier detection algorithms have 4 basic parameters:

- k - neighbours count threshold;
- R - distance threshold;
- W - window size;
- S - slide size.

Usually they are set for each dataset separately according to the specifics of the dataset, memory facilities and the stream of the data. The results in this section will be presented for each dataset separately when varying values of parameters.

Parameters W and S determine the volume and the speed of the data streams. They are the major factors that mostly affect the speed and memory usage of the algorithms. Values k and R determine the outlier rate and mostly affects outlier detection accuracy. Memory consumption is also related to k as all the algorithms store information regarding k neighbours of each data point.

In this research we will analyse what is the influence of changing parameter values for distance-based outlier detection algorithms when working with stream data. To clearly see the impact of exact parameter other parameters will be fixed.

From the literature we found that usually window size is set quite big (f. ex. $W = 10000$ or $W = 100000$ [4]) but in some cases the smaller window is used (e.g. $W = 400$ or $W = 1000$ [12]). If we take smaller window our algorithm will run faster because less data points need to be processed in a window. Since our goal is working on algorithm development for faster execution we will consider a case when window size $W = 500$ and slide size $S = 500$ unless stated otherwise (which means that each data point is processed only in one window).

On the other hand parameters R and k needs to be set for each dataset separately because the choice of these parameters is strongly related to the dataset specifics, outlier rate and also chosen W. By experimenting while changing k and R best match of parameters was chosen. We will consider them as default parameters. These parameters are presented in Table 4.

Further the research on changing values of parameters is presented. We will start from varying k and R and prove the choice of default parameters. Then we will move on varying W and S and see if the change of window size and slide size influence the outlier detection results. Each case will be discussed in separate subsections.

Table 4: Default parameters R and k

Dataset	k	R
Shuttle	55	0.25
KDD Cup 1999	5	0.35
Pendigits	15	1.10
Forest Cover	10	0.40

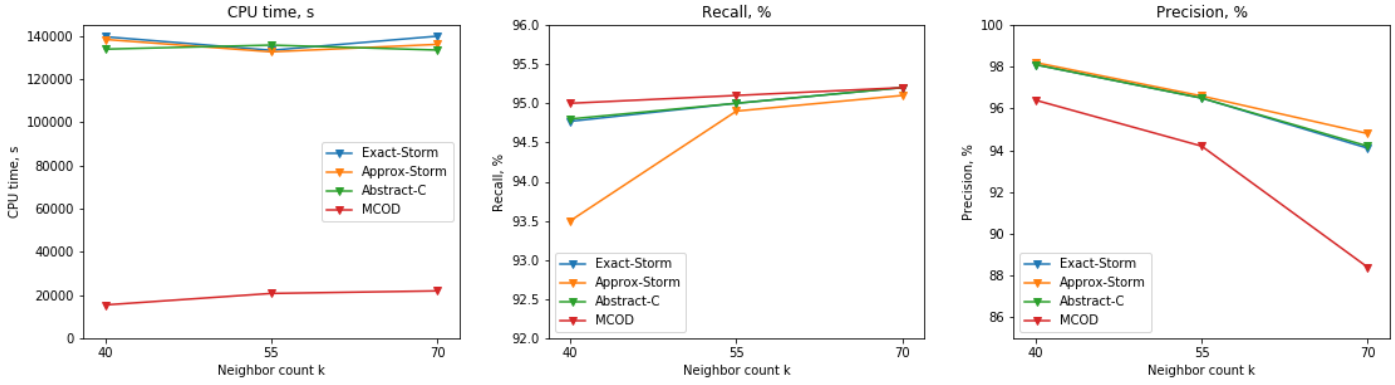
2.2.1. Varying parameter k

Parameter k defines how many neighbours are required within assigned distance R so we could call a data point inlier. Choice of k mostly affects outlier detection accuracy. More data points are reported as outliers if we increase k and less data points are reported as outliers if we decrease k. To detect outliers accurately we need to find optimal k which gives high recall and high precision. However, usually when recall increases precision decrease and vice versa.

Let's discuss results for each dataset separately when varying k. In this experiment parameters R, W and S are fixed (considered default parameters).

Note: pay attention that scales in the graphs may differ. Also in some graphs not all algorithm curves are visible, it implies that hidden algorithms behave same way as other algorithms in the graph.

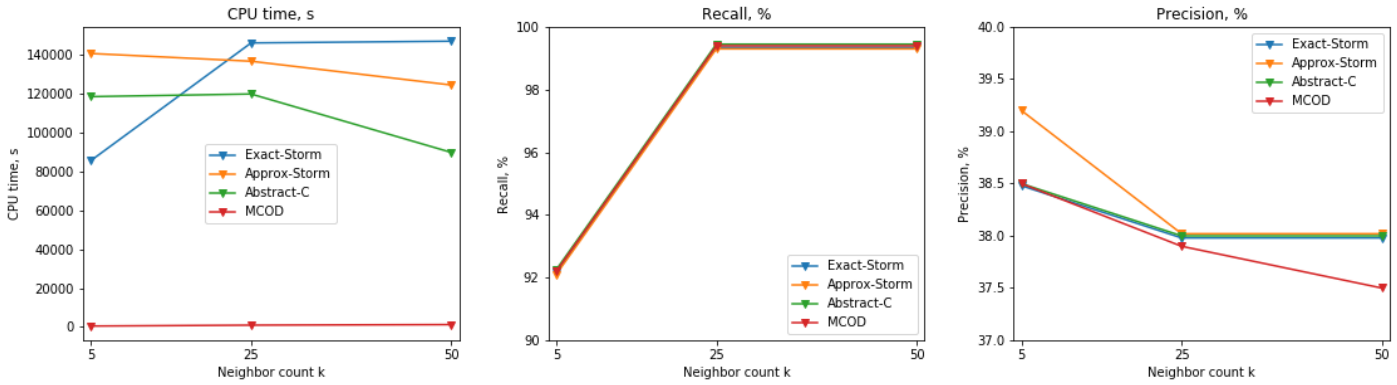
Figure 7: Shuttle dataset: how changes time and accuracy when varying neighbour count k



Shuttle dataset stands out from other datasets with high outlier detection recall and precision. Figure 7 shows how change of k influences outlier detection processing time and accuracy. If we look to the 2nd and 3rd graphs we can prove that when we vary k if recall increase then precision decrease and vice versa. These observations state that varying k has significant impact on algorithm accuracy. On the other hand, processing time is weakly affected by change of k (1st graph). Also it is worth to highlight how relatively fast MCOD works with Shuttle dataset. The main reason behind this is that outliers easi-

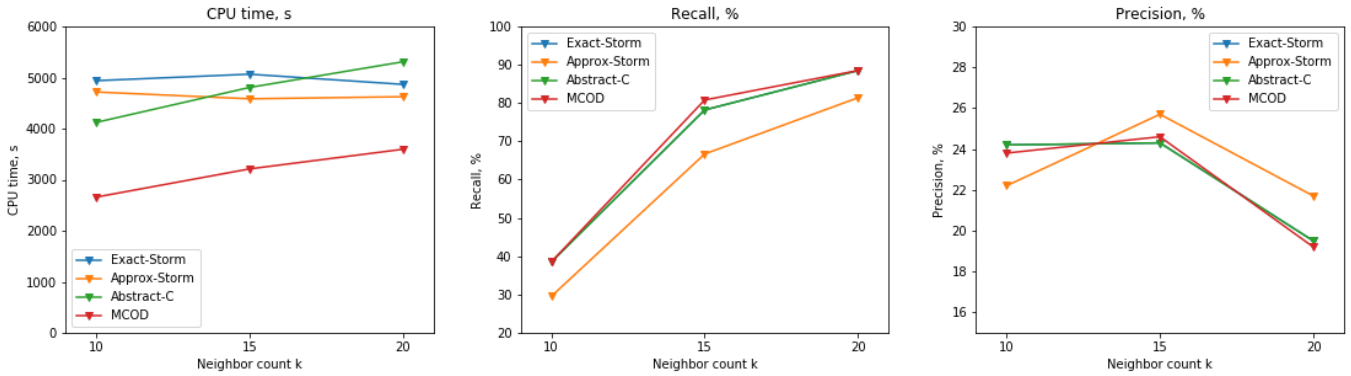
ly differs from the rest of the data and when that happens while using MCOD algorithm, keeping neighbourship information in micro-clusters significantly lowers processing time.

Figure 8: KDD Cup dataset: how changes time and accuracy when varying k



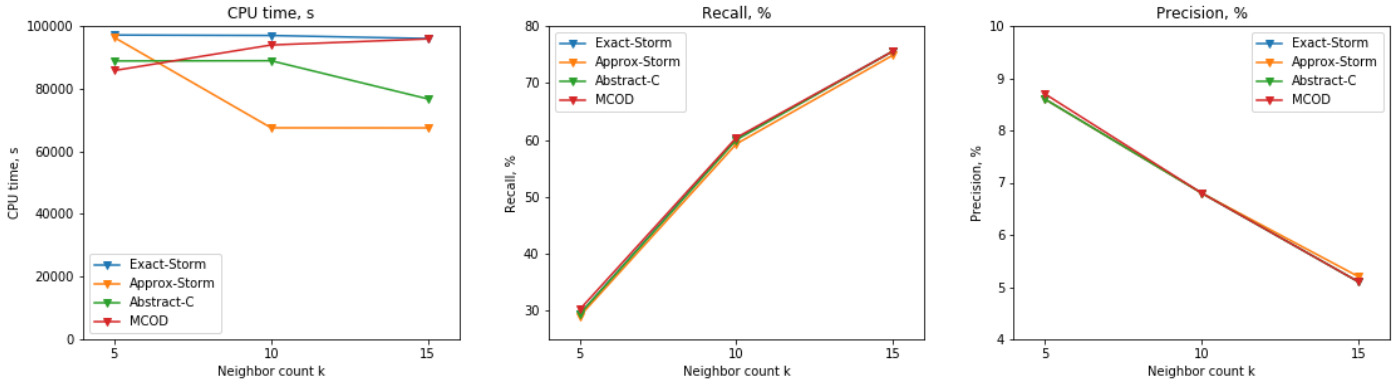
Influence of varying k for KDD Cup 1999 dataset is illustrated in Figure 8. This dataset is an example where using distance-based outlier detection algorithms for stream data gives high recall and low precision (means that labeled outliers are well recognized but as well many false outliers are reported). Let's see if increased k can add value to the accuracy. From the 2nd graph we see that increasing neighbours count from 5 to 25 for KDD Cup 1999 dataset gives an increment in recall approximately by 7% and is close to perfect result. Further incrementation of k is not that significant. However, incrementing k provokes decrease in precision (3rd graph). Talking about algorithm processing time, once more MCOD stands out with low CPU time while other algorithms is noticeably affected by changing k. CPU time for Exact-Storm increases when we increment k while for Approx-Storm and Abstract-C slightly goes down. The main reason behind this is that Exact-Storm algorithm keeps in memory list of k most recent preceding neighbours and with bigger k working with this list causes increased CPU time.

Figure 9: Pendigits dataset: how changes time and accuracy with different k



Applying distance-based outlier detection algorithms on Pendigits dataset let us reach recall around 80% with precision around 25%. Figure 9 shows how change of k influences outlier detection algorithm processing time and accuracy. If we look at 2nd and 3rd graphs once more we see that if we increase k, recall increase and precision decrease (with exception of Approx-Storm algorithm where applied approximations gives little bit higher precision when $k = 15$). Processing time in this case is insignificantly affected by change of k, only Abstract-C algorithm shows some ascendant trend in CPU time (1st graph). Again MCOD is relatively faster when comparing to other algorithms.

Figure 10: Forest Cover dataset: how changes time and accuracy with different k



Impact of varying k for Forest Cover dataset is shown in Figure 10. For this dataset distance-based outlier detection algorithms do not give great results: recall reaches 60-70% while precision is very low. This is because outliers are not noticeably separable from other data (outliers can be recognized only with changes observed in few features of a data point). CPU time for Exact-Storm algorithm keeps in the same level when changing k while other algorithms give some noticeable reaction. MCOD CPU time increases when we increase k because more data points fall into PD list and not micro-clusters and that increases number of distance computations. On the other hand CPU time for Abstract-C and Approx-Storm algorithms decreases as in the example with KDD Cup 1999 dataset. From the 2nd graph we see that increasing neighbours count significantly increases recall but precision gradually shrinks (3rd graph).

To generalize, to achieve best result parameter k needs to be adjusted to each dataset separately according to the dataset specifics, outlier rate and chosen outlier detection algorithm parameters. Common behaviour when we vary neighbour count k is that once we increase k we can increase recall but then precision goes down and vice versa. Speaking about the processing time, change of k does not significantly influence CPU time. From this experiment we saw that accuracy received with algorithms Exact-Storm, Approx-Storm and MCOD is quite similar while for Approx-Storm little differs. Talking about processing time MCOD algorithm stands out with low CPU time for 3 out of 4 datasets.

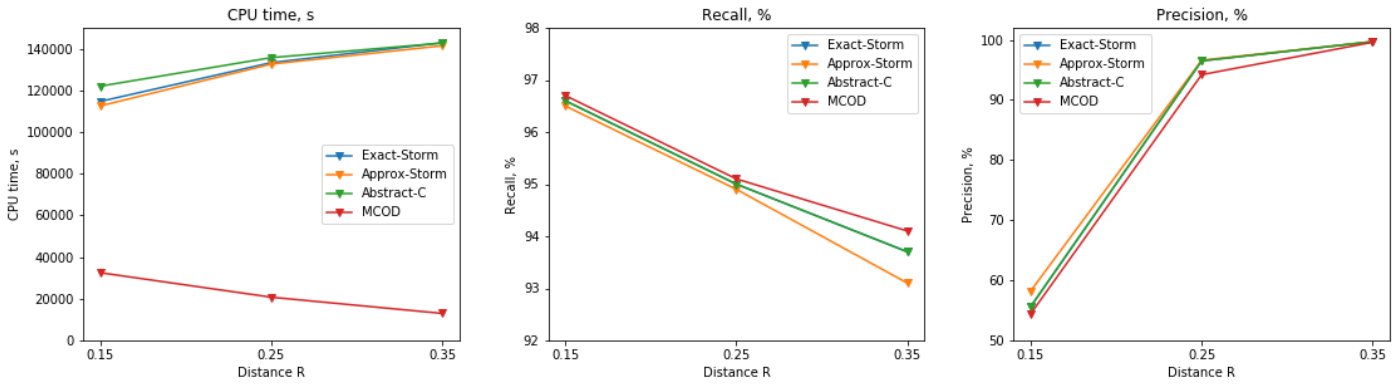
2.2.2. Varying parameter R

Parameter R defines the distance in which we look for k neighbours for specific data point so we could call it an inlier or outlier. Choice of R mostly affects outlier detection accuracy. It works in an opposite way compared to varying another parameter k. More data points are reported as outliers if we decrease R and less data points are reported as outliers if we increase R. Usually when recall increase precision decrease and vice versa, so to detect outliers accurately we need to find optimal R which gives high recall with high precision.

Further we will discuss results for each dataset separately when varying parameter values R. In this experiment k, W and S values are fixed.

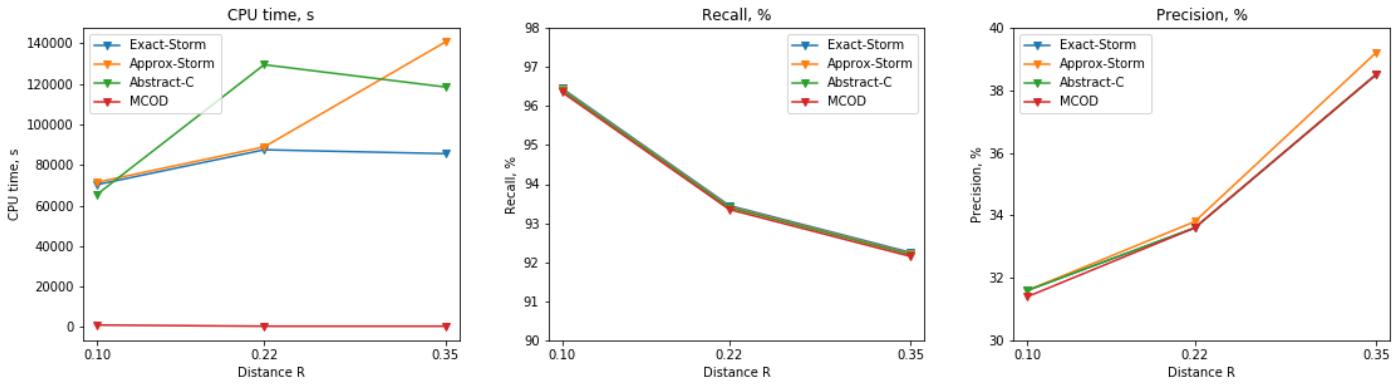
Influence of varying R for Shuttle dataset is illustrated in Figure 11. When we increase the distance R we observe a continually declining recall (2nd graph) and increasing precision (3rd graph). This happens because increased R means that we search for neighbours within wider radius from the data point and as a result less data points are reported as outliers. Talking about CPU time, since less data points are considered as outliers in MCOD algorithm more data points are kept in micro-clusters so CPU time consumption decrease. For other algorithms it works in opposite way - more data points have neighbourship information and working with it increases CPU time.

Figure 11: Shuttle dataset: how changes time and accuracy when varying distance R



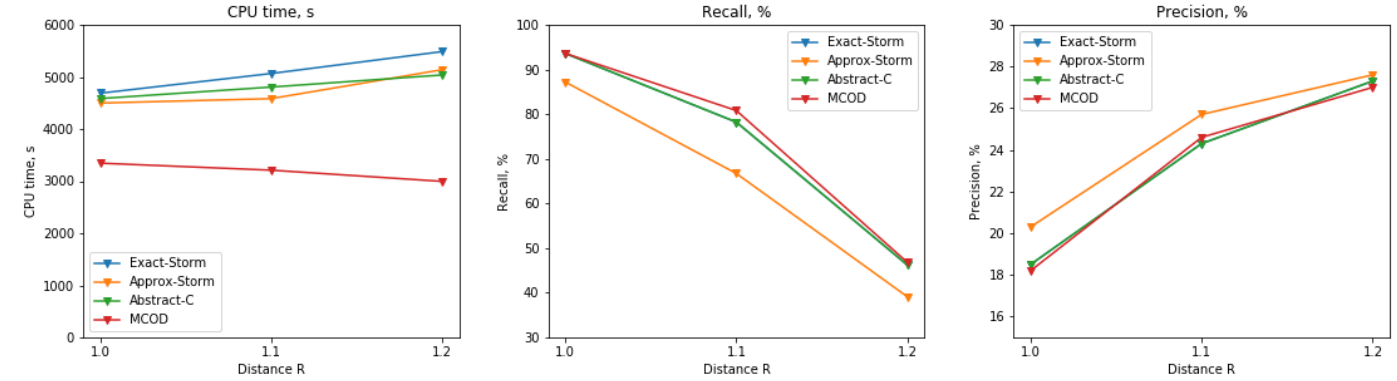
After applying distance-based outlier detection algorithms on KDD Cup 1999 dataset with varying R we can see some similar trends. The graphs for this dataset are presented in Figure 12. Processing time in this case is noticeably affected by change of R (1st graph) but overall tendencies remain similar as with Shuttle dataset. Again MCOD is relatively faster and other algorithms perform some increase in CPU time with incremented R. In 2nd and 3rd graphs we can check how change of R influences outlier detection accuracy. From the graphs we see that increased R causes recall decline and precision grow. Accuracy among all 4 algorithms with KDD Cup 1999 is very similar.

Figure 12: KDD Cup 1999 dataset: how changes time and accuracy with different R



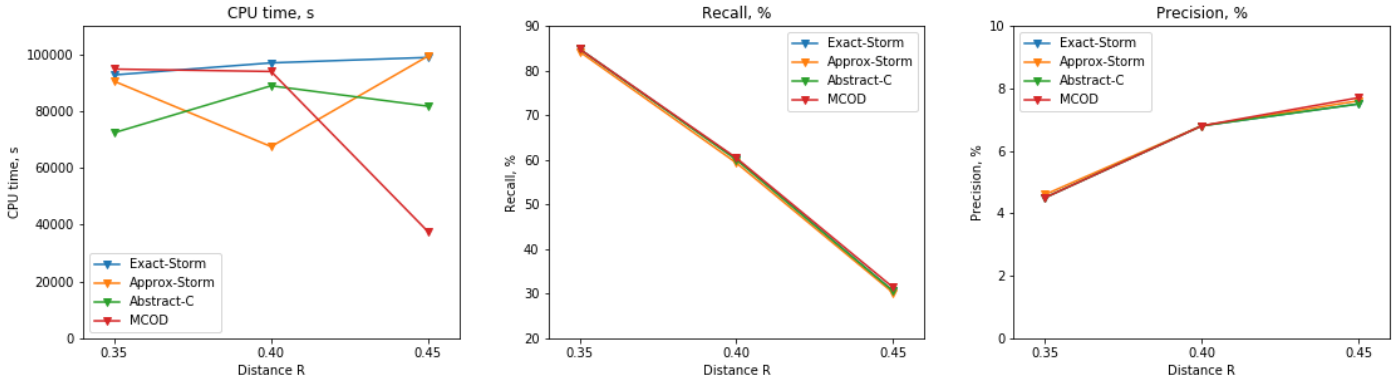
Pendigits dataset also behaves very similarly as two already described datasets. Corresponding graphs are presented in Figure 13. As talked before varying parameter R has a little impact on CPU time. To be precise MCOD algorithm has a slight decrease in processing time while Exact-Storm and Abstract-C a little increase (1st graph). Speaking about accuracy with Pendigits dataset remains the same tendency: increased R provokes decline in recall and increase in precision. However, while recall shows acceptable results precision is comparably low.

Figure 13: Pendigits dataset: how changes time and accuracy when varying distance R



Influence of varying R for Forest Cover dataset is shown in Figure 14. When we increase the distance R we observe a continuous decrease in recall (2nd graph) and slight increase in precision (3rd graph). Talking about CPU time for algorithm Exact-Storm it stays quite stable when changing R. MCOD in contrast with higher R shows a significant drop in CPU time caused by situation that more data points are in micro-clusters. Approx-Storm as well as Abstract-C show some fluctuations caused by changing distance R.

Figure 14: Forest Cover dataset: how changes time and accuracy when varying distance R



To generalize, parameter R needs to be adjusted to each dataset separately according to the dataset specifics, outlier rate and other chosen parameters. Common behaviour when we vary distance R is that once we increase R the recall decreases but then the precision goes up and vice versa. The goal working with a specific dataset is to find best case where recall is highest with high precision. Talking about CPU time from this experiment we can state that when increasing R processing time for MCOD algorithm tends to decrease and for other algorithms slightly increase or stay stable.

2.2.3. Varying parameter W

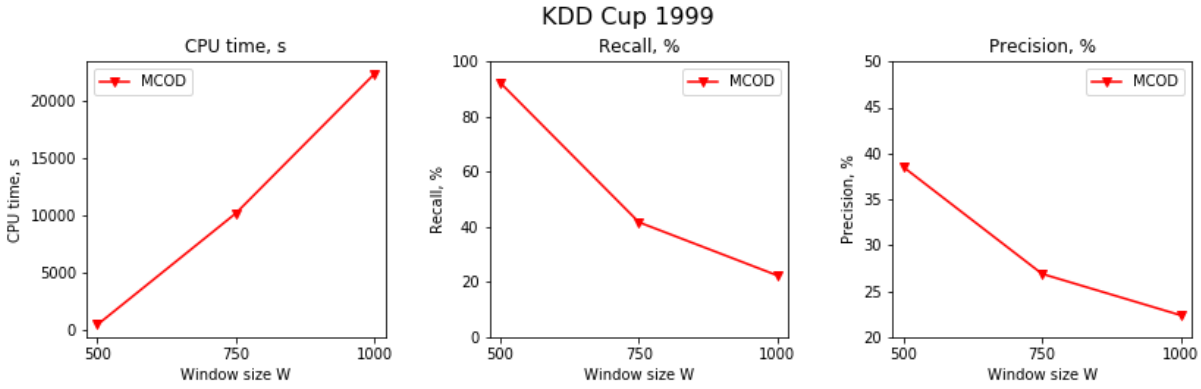
In this subsection influence of varying window size W is presented. We will discuss how changes time consumption, recall and precision of distance-based outlier detection algorithms when we are increasing window size from 500 to 1000 data points a window.

The parameters S and R are fixed for all the cases while parameter k is adjusted according to window size W. In previous experiments 3 out of 4 parameters were fixed but this time we decided to adjust 3rd parameter k when changing window size.

Let's discuss Figure 15 which illustrates case where MCOD was applied to KDD Cup 1999 dataset by fixing all 3 parameters and varying only window size. If we increase window size W and fix the neighbour count k then we get constantly decreasing recall and precision. It might be explained by the fact that with increasing number of data points in a window the number of neighbours for each data point also will increase. What follows next is that less data points will be considered as outliers and we will have declining algorithm accuracy.

Keeping in mind highlighted trend it is more interesting to investigate what happens if we increase window size W and respectively increase k and check the hypothesis that once we find appropriate neighbour count threshold k, window size does not play significant role on the accuracy of the outlier detection algorithms. For further experiment when

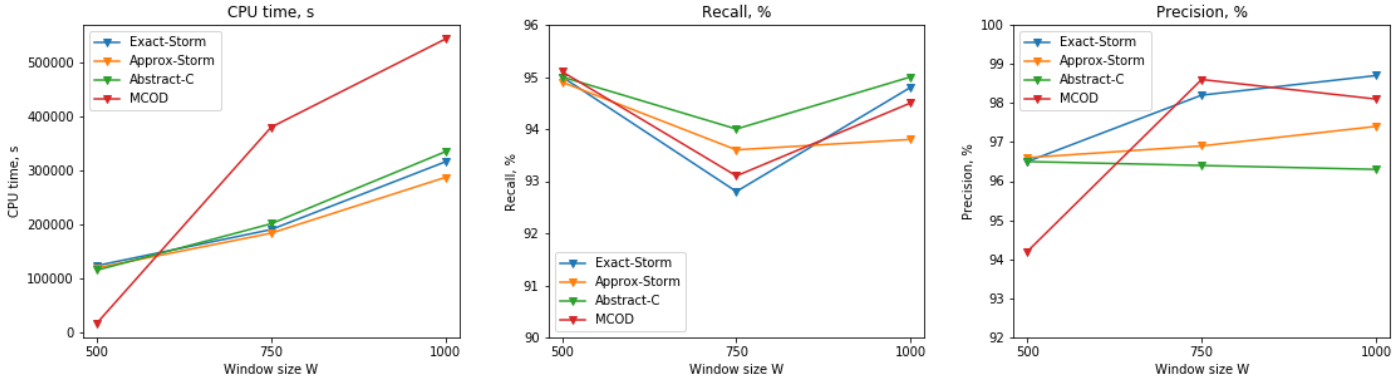
Figure 15: KDD Cup 1999: accuracy when varying window size W



$W = 500$ we choose default parameter k (described in the Table 4), for $W = 750$ and $W = 1000$ default parameter k is increased by 1.5 or 2 times respectively.

Figure 16 illustrates what happens once we increase parameter W (with increasing parameter k respectively) for dataset Shuttle. From the 1st graph we see how CPU time increases with W. The main reason of incrementing CPU time is that for bigger window we need more distance computations between data points than in a smaller window. What stands out in this graph is that CPU time for MCOD algorithm unexpectedly increases when $W = 750$. The possible explanation is the specifics of the dataset. Since we increased parameter k more data points were assigned to PD list and more distance computations were needed. If we look at the graphs of accuracy there are some fluctuations but no significant difference with varying windows size W.

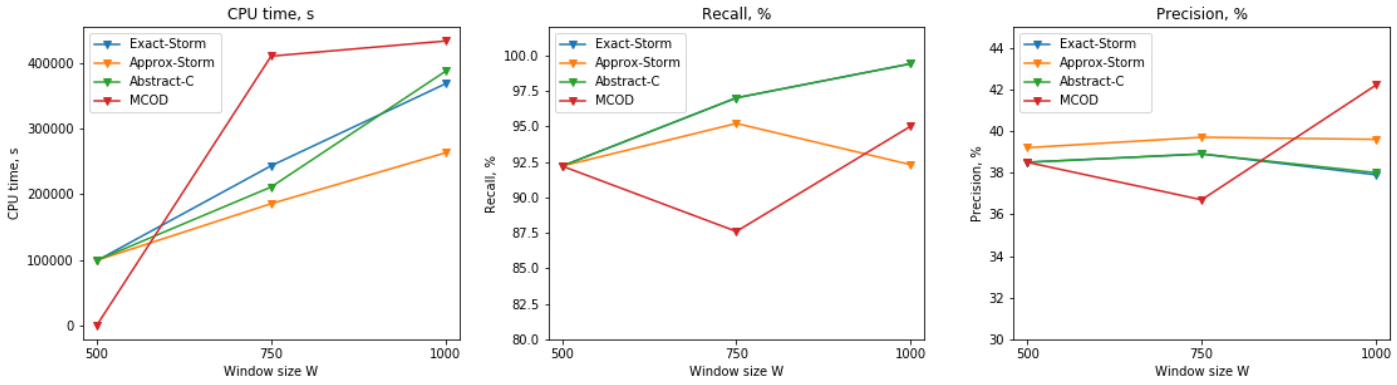
Figure 16: Shuttle dataset: how changes time and accuracy when varying window size W



Influence of varying W for KDD Cup 1999 dataset is shown in Figure 17. When we increase the window size W we observe an increase in CPU time (1st graph). Once again MCOD algorithm proves to be faster when we have $W = 500$ and increases rapidly with increased window size. Talking about recall and precision it behaves differently with

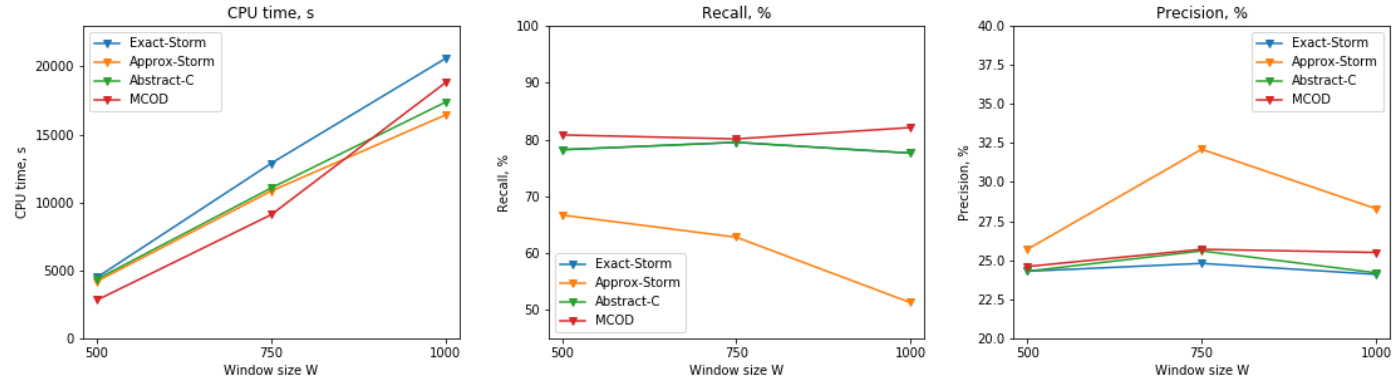
separate algorithms. For example, Abstract-C and Exact-Storm with KDD Cup 1999 dataset show better results with bigger window than with smaller. In contrast other algorithms show some fluctuations. However, it seems that these fluctuations are mostly related to the specifics of the dataset because they are only observed with this dataset.

Figure 17: KDD Cup 1999 dataset: change of time and accuracy with different W



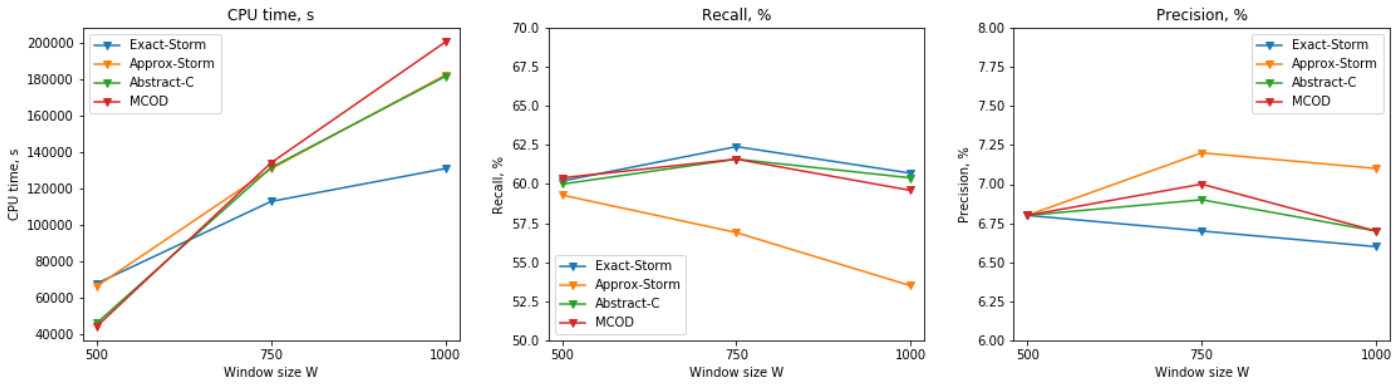
Pendigits and Forest Cover datasets are illustrated in Figure 18 and Figure 19. These datasets behave similarly when we change W so we will generalize them together. CPU time is increasing continuously together with increased window size and does not show any significant differences among algorithms. From the accuracy graphs we can state that increased W has no noticeable impact for algorithms MCOD, Abstract-C and Exact-Storm if we increase k respectively to the window size. On the other hand Approx-Storm has declining recall and slightly increasing precision when changing W.

Figure 18: Pendigits dataset: how changes time and accuracy when varying window size W



To generalize, if we adjust parameter k respectively, changing parameter W does not have significant impact on overall outlier detection accuracy. What changes when we increase W is that CPU time increases constantly together with increased W.

Figure 19: Forest Cover dataset: how changes time and accuracy with different W



Another thing that we observed with this experiment is that MCOD strongly reacts to the change of W and k because micro-clusters consider area with radius $R/2$. It means that for MCOD if we want to increase W we should reconsider parameters k and R and find ones which uses the advantage of MCOD algorithm. Also it is worth to mention that Approx-Storm because of used approximations noticeably reacts to varying W.

2.2.4. Varying parameter S

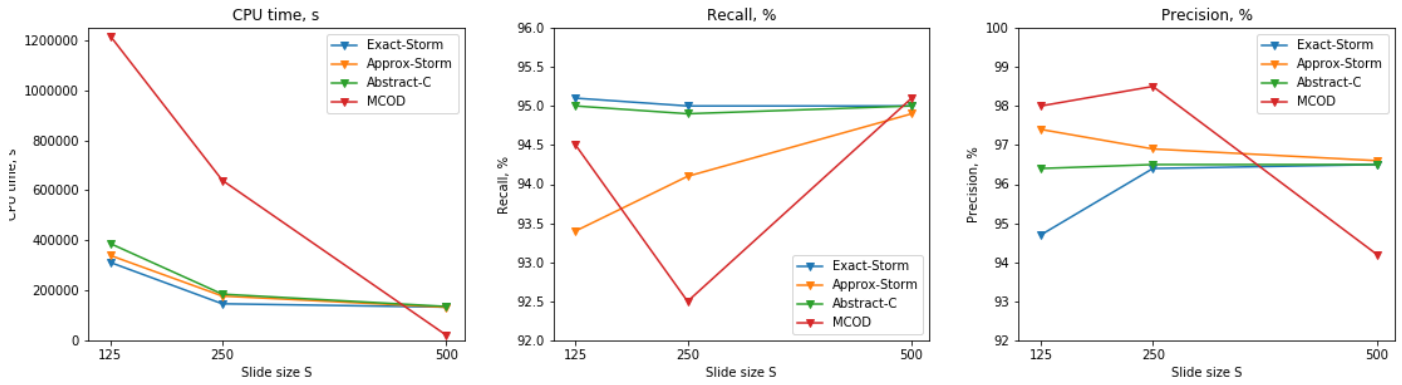
Parameter S defines the step by which sliding window is processed. By choosing S we can control the speed of outlier detection algorithm and the number of windows in which one data point will be processed.

Basic rule is that reducing slide size S increases processing time because this way we need to work with more windows. However the main question is how does it influence outlier detection accuracy because the goal is to find optimal situation when we have high accuracy with the shorter algorithm processing time. This is what we seek to answer in this subsection.

For this experiment we have fixed parameters k, R and W, and varying slide size S. Let's see what observations can be done after investigating 4 datasets.

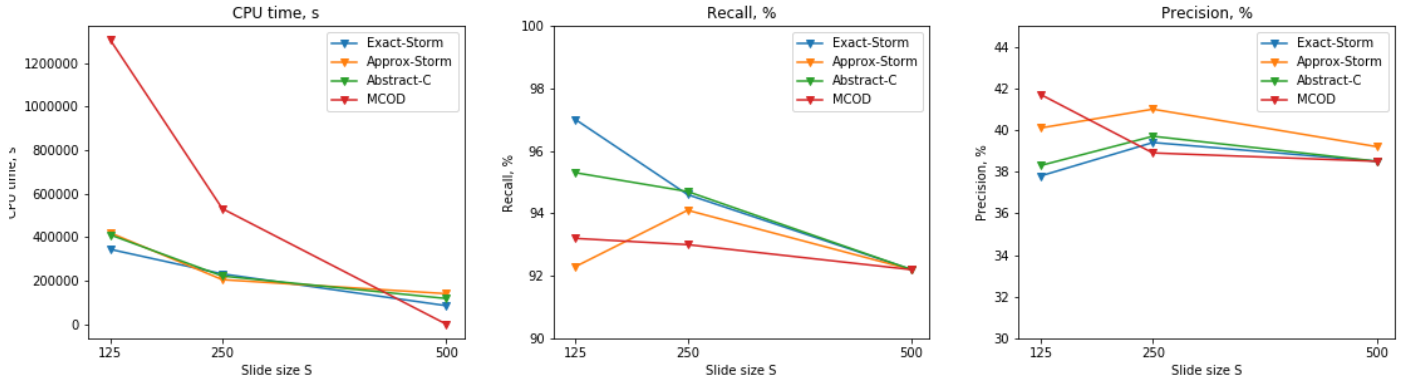
Shuttle dataset for which distance-based outlier detection algorithms work best is presented in Figure 20. As expected if we take smaller slide size S CPU time increases no matter which algorithm is considered (1st graph). MCOD algorithm stands out if we consider CPU time because of micro-clusters eliminations and creations. In some cases after deleting expired objects we may need to eliminate the cluster and after new data points arrive create a micro cluster. This process might have an impact to increasing MCOD CPU time when we take smaller S. If we talk about the accuracy slide size has little influence on recall and precision for a Shuttle dataset.

Figure 20: Shuttle dataset: how changes time and accuracy when varying slide size S



Varying slide size S for another dataset KDD Cup 1999 is presented in Figure 21. CPU time (1st graph) shows same tendencies that smaller S increases outlier detection algorithms processing time. Looking into the accuracy (2nd and 3rd graphs) there is some fluctuation in recall and precision but no major impact. However from the graphs we can state that for KDD Cup 1999 dataset the case where $S = 250$ which means that each data point is considered in two windows works best.

Figure 21: KDD Cup 1999 dataset: change of time and accuracy when varying slide size S



The datasets Pendigits and Forest Cover and their reaction to varying S is illustrated in Figure 22 and Figure 23. As expected CPU time is bigger with smaller slide size S and MCOD again have significantly higher processing time with $S = 125$. Accuracy on the other hand stays quite stable with different slide size. Approx-Storm algorithm has relatively lower recall and higher precision if we compare to other algorithms because of implemented approximations in outlier detection process.

To generalize, slide size S defines the speed of the outlier detection algorithm and the number of windows in which one data point will be included. The experiment justify

Figure 22: Pendigits dataset: how changes time and accuracy when varying slide size S

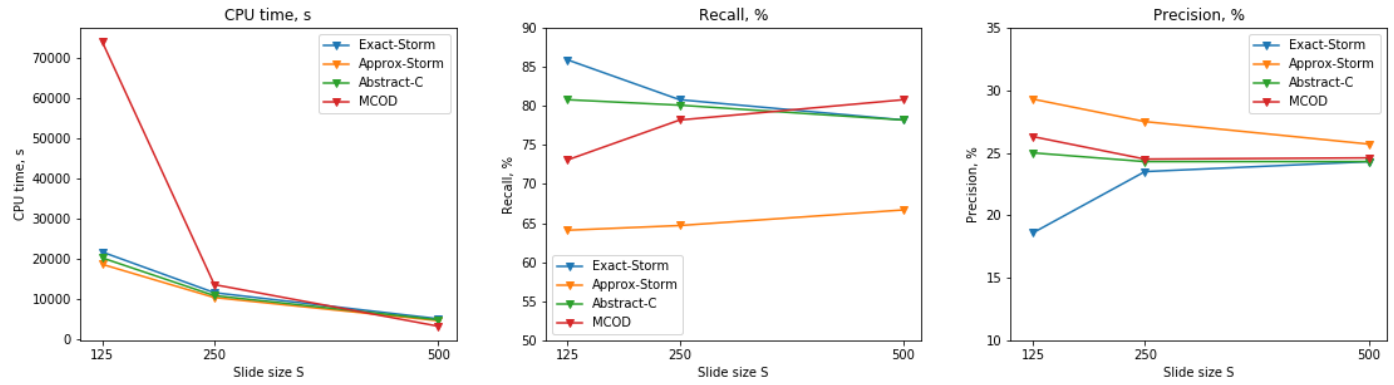
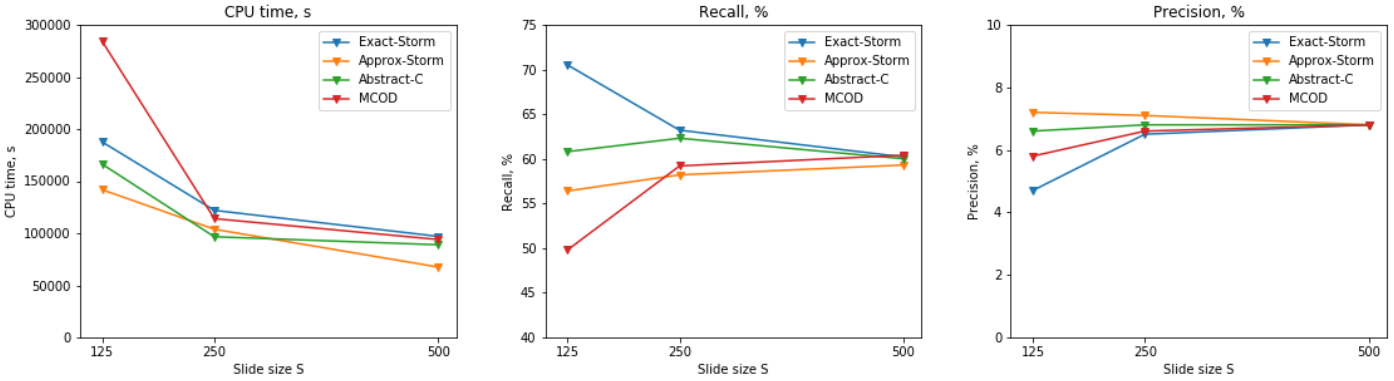


Figure 23: Forest Cover dataset: how changes time and accuracy when varying slide size S



that smaller S increases CPU time. Moreover we found that choice of S does not show significant impact on outlier detection accuracy. Speaking about algorithm performance Approx-Storm stands out in a way that approximations cause slightly lower recall no matter S and MCOD has significantly higher processing time with smaller S.

Summing up, the Section 2.2 consists of analysing choice of distance-based outlier detection algorithm parameters and its influence to the algorithm performance. Overall we investigated 4 parameters: k, R, W and S. These experiments let us come to some basic conclusions about the choice of parameters.

Parameter k defines the threshold of number of neighbours in a window needed to declare if data point is an inlier or outlier. We came up to the conclusion that for different datasets k needs to be adjusted individually because of the different dataset characteristics. Also we identified a common behaviour that increasing k value let us increase the outlier detection recall but then the precision decreases. Speaking about algorithm processing time choice of k has no significant influence to the CPU time.

Another parameter R defines the distance which determine the area around the data point in which we look for its neighbours. What we found out is that if we increase distance R the recall tends to decrease while the precision increase. This might be explained by the fact that with bigger R more data points have more than k neighbours and number of reported outliers declines.

Afterwards we investigated few different window sizes and how the choice of W impacts the result if we fix parameters R and S, and adjust k according to the window size. The experiment showed that if we adjust k, different window sizes have very small or no impact on outlier detection accuracy. What changes when we increase W is that algorithm processing time continuously geos up.

Lastly we considered the change of parameter S - slide size by which we move the sliding window after one is processed. We found out that it mostly influences algorithm processing time but has little impact on outlier detection accuracy.

Table 5: Results of distance-based outlier detection algorithms application on 4 datasets

Algorithm	Peak Memory, KB	CPU Time, s	Recall	Precision	Outlier rate*
Shuttle dataset ($W = 500$, $S = 500$, $k = 55$, $R = 0.25$)					
Exact-Storm	87744	123854	95.0%	96.5%	7.03%
Approx-Storm	98008	119643	94.9%	96.6%	7.02%
Abstract-C	87528	115414	95.0%	96.5%	7.03%
MCOD	90596	17043	95.1%	94.2%	7.22%
KDD Cup 1999 dataset ($W = 500$, $S = 500$, $k = 5$, $R = 0.35$)					
Exact-Storm	11175	98724	92.2%	38.5%	0.80%
Approx-Storm	98840	99277	92.2%	39.2%	0.79%
Abstract-C	97972	99292	92.2%	38.5%	0.80%
MCOD	124640	387	92.2%	38.5%	0.80%
Pendigits dataset ($W = 500$, $S = 500$, $k = 15$, $R = 1.1$)					
Exact-Storm	79056	4486	78.2%	24.3%	7.32%
Approx-Storm	89132	4198	66.7%	25.7%	5.90%
Abstract-C	78548	4354	78.2%	24.3%	7.32%
MCOD	78084	2819	80.8%	24.6%	7.45%
Forest Cover dataset ($W = 500$, $S = 500$, $k = 10$, $R = 0.4$)					
Exact-Storm	688508	96952	60.2%	6.8%	8.64%
Approx-Storm	668472	67491	59.3%	6.8%	8.48%
Abstract-C	750152	104417	60.0%	6.8%	8.64%
MCOD	991120	93924	60.4%	6.8%	8.66%

* Detected outlier rate shows what percentage of the dataset is reported as outliers

All experiments performed in Section 2.2 showed how important is choosing the right parameters in distance-based outlier detection algorithms when working with stream data. When choosing the parameters main goal is to find optimal solution which would bring highest accuracy with considerably low processing time. In Table 5 we give the results of

algorithms Exact-Storm, Approx-Storm, Abstract-C and MCOD applied to the 4 selected datasets with optimal parameters (the cases where $W = 500$ and S=500 are considered).

In Table 5 there is an additional column of detected outlier rate which indicates what part of the dataset is predicted to be outliers. Mostly we see that if precision and recall is high, the predicted outlier rate is very close to the real one which was given in Table 3 in Section 2.1. If we have low precision which means that many false outliers are predicted then naturally we have higher than the real outlier rate.

Speaking about algorithms performance, in respect to outlier detection accuracy all of them are very similar and this is because the same distance-based approach is used. What needs to be highlighted according to the accuracy is Approx-Storm algorithm. Its recall and precision deviates from other algorithms the most and this is because of approximations usage. Speaking about algorithms processing time, mostly stands out MCOD: in cases where outliers can be separated more easily MCOD is significantly faster but on opposite situations or with wrongly chosen parameters its processing time can vastly exceed a usual CPU time. Also I want to mention algorithm Abstract-C. It does not show much better accuracy than others or does not significantly stands out with respect to CPU time but it is most stable among analysed algorithms.

In the next subsection we will have another part of experiments which help to investigate if suggested distance-based outlier detection algorithms improvements contribute to the better outlier detection accuracy and overall algorithm work. Results received using modified algorithms will be compared to those presented in Table 5.

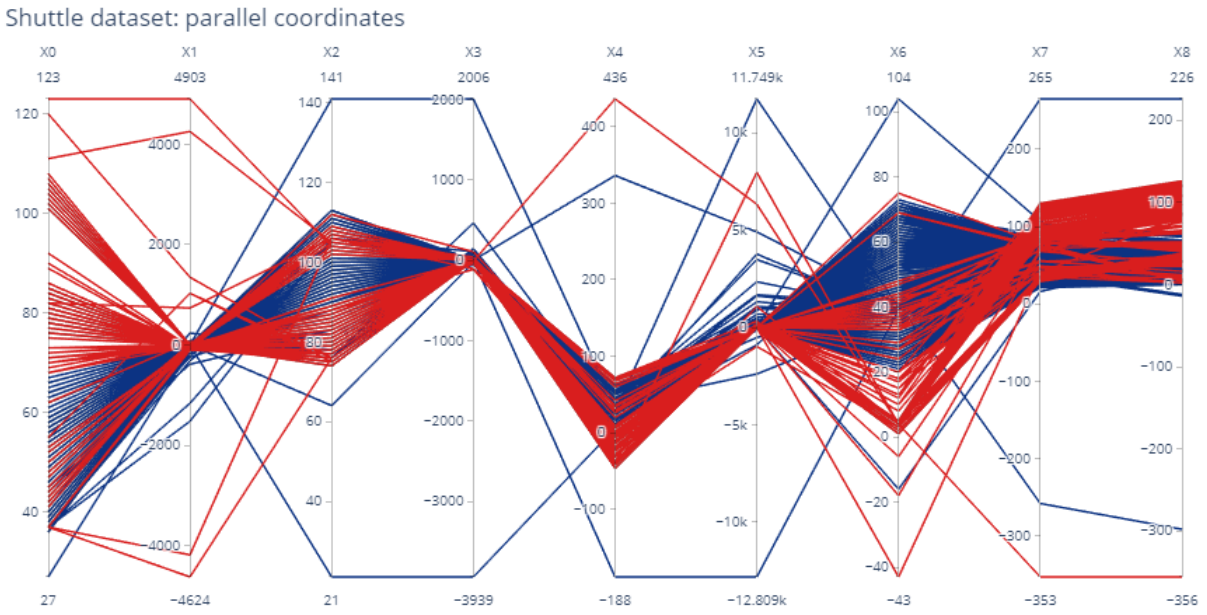
2.3. Using SVM for algorithms improvement

In this section we will discuss the results of using SVM for improving distance-based outlier detection algorithms. In Section 1.3 modifications related to SVM were presented: usage of linear SVM coefficients for weighted distance and applying SVM for verify if outliers do not behave as an inliers when their behaviour is learnt from training subset. For latter modification two cases: linear and non-linear SVM was considered. These modifications will be applied separately and results for each dataset presented in tables.

Let's discuss how SVM was trained. SVM for datasets Shuttle and Pendigits were trained on 1000 data points while for KDD Cup 1999 and Forest Cover on 3000 data points (more points were used because of small outlier rate in these datasets).

One of the modifications is using linear SVM coefficients as weights to define weighted distance which will be used in distance-based algorithms. Let's talk about one example and see what coefficients are assigned to the features.

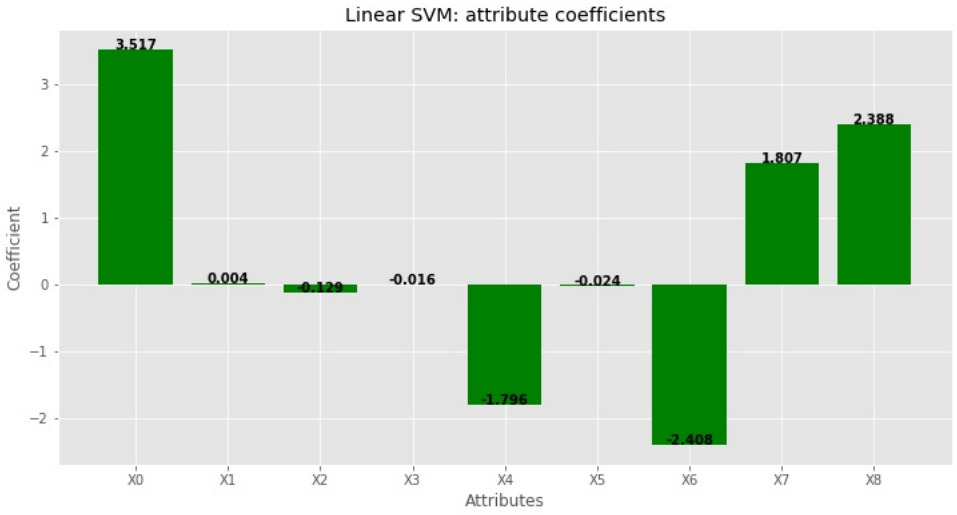
Figure 24: Shuttle: Dataset visualization (10000 points). Red lines are data points that are labeled as outliers.



If we compare Shuttle dataset visualization (Figure 24) to assigned linear SVM coefficients (Figure 25) we can see some common tendencies. Let's consider features x_0 and x_1. In Figure 24 we can see that labeled outliers (red lines) tend to have higher values of feature x_0 than other data points (blue lines). This indicates that feature x_0 is rather important for identifying outliers and this is why the first coefficient of linear SVM has a high value of $|w_0| = 3.517$. Opposite situation is with feature x_1. From Figure 24 we see that both classes (outliers and not outliers) have quite similar values for attribute x_1 and also the coefficient of linear SVM is rather small with a value $|w_1| = 0.004$.

These discussed graphs show that labeled outliers in Shuttle dataset tend to differ with

Figure 25: Shuttle: coefficients of linear SVM



respect to features x_0, x_4, x_6, x_7 and x_8. This is valuable information and it will be used in distance-based outlier detection algorithms.

To see what information from training samples about the feature importance was learnt for other datasets, you can find the visualizations of datasets as well as linear SVM coe cients in Appendix.

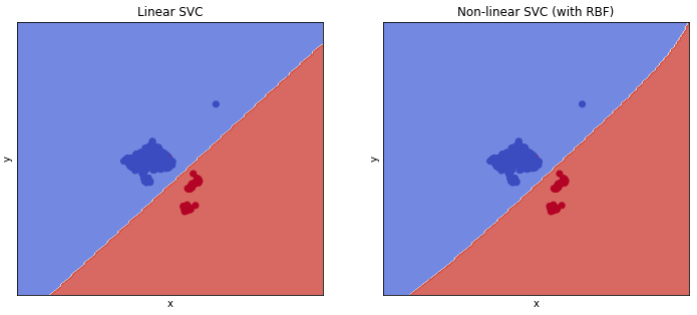


Figure 26: Shuttle: trained SVM

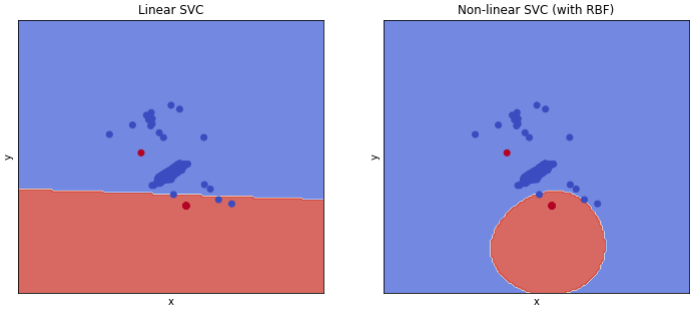


Figure 27: KDD Cup: trained SVM

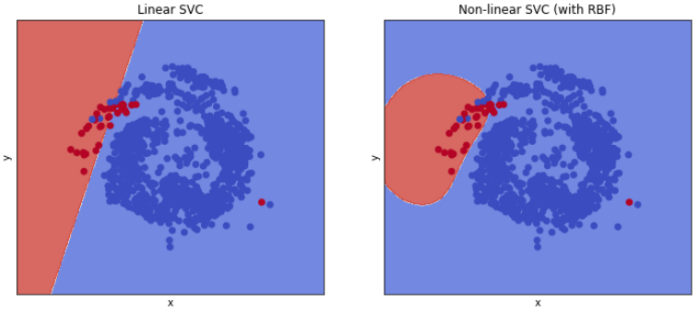


Figure 28: Pendigits: trained SVM

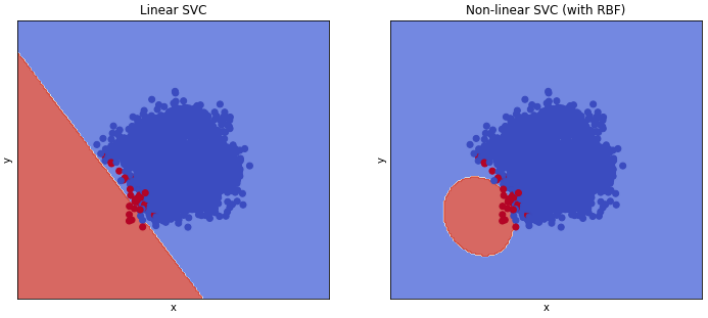


Figure 29: Forest Cover: trained SVM

The idea of another modification is that there exist some visible outlier patterns but distance-based outlier detection does not catch them because all they rely on is distances

between the data points. So what we decided to do is retrieve useful information from the training sample with labeled outliers and use together with distance-based outlier detection algorithms.

As we said before, SVM for datasets Shuttle and Pendigits were trained on 1000 data points while for KDD Cup 1999 and Forest Cover 3000 data points were used. Figures 26-29 show how linear and non-linear (using RBF function) SVM was trained on subsets of our datasets. From these graphs we can see the behaviour of labeled outliers (red colour) and how SVM learnt to separate inliers and outliers. We can see that Shuttle dataset is the one where outlier class separates easily and for Pendigits and Forest Cover datasets outliers and inliers form one cloud which makes distance-based outlier detection very complex. Fortunately we see that SVM is able to generalize outliers' behaviour. Let's see if this information helps improving outlier detection accuracy.

To separate modifications we will mark them with * respectively, e.g. MCOD algorithm which will use linear SVM weights for weighted distance will be marked as MCOD*, algorithm where linear SVM will be used as additional filtering will be marked as MCOD** and algorithm where non-linear SVM will be used as additional filtering will be marked as MCOD***.

The experimental results of proposed modified algorithms are presented in Tables 6-9 for each dataset separately.

Table 6: Shuttle dataset: Results of algorithms modifications

Algorithm	Peak Memory, KB	CPU Time, s	Recall	Precision
Exact-Storm	87744	123854	95.0%	96.5%
Exact-Storm*	124916	99334	95.0%	98.6%
Exact-Storm**	107740	121056	94.2%	99.9%
Exact-Storm***	108604	117810	94.0%	100.0%
Approx-Storm	98008	119643	94.9%	96.6%
Approx-Storm*	125684	100356	95.0%	98.7%
Approx-Storm**	121000	116467	94.2%	99.9%
Approx-Storm***	109828	110507	94.0%	100.0%
Abstract-C	87528	115414	95.0%	96.5%
Abstract-C*	124916	97278	95.0%	98.6%
Abstract-C**	117704	118398	94.2%	99.9%
Abstract-C***	119428	114816	94.0%	100.0%
MCOD	90596	17043	95.1%	94.2%
MCOD*	127524	9678	95.0%	96.5%
MCOD**	123788	13284	94.2%	99.9%
MCOD***	123852	10425	94.0%	100.0%

* implementing weighted distance
** modification of using linear SVM to double check outlierness
*** modification of using non-linear SVM to double check outlierness

With Shuttle dataset (Table 6) classical distance-based outlier detection algorithms show quite good results - recall and precision varies around 94-97%. From the experiments we can state that introducing weighted distance where weights are taken from trained linear SVM slightly increases precision while recall stays very similar. On the other hand using linear and non-linear SVM as an additional step for outlier verification increases precision to 99.9-100.0%, however this modification causes a small decrease in a recall.

Surprisingly almost all introduced modifications with Shuttle dataset caused an algorithm processing time reduction. What happened is that introduced weighted distance gathers even closer similar data points and inliers are detected easier which fastens outlier detection. On the contrary this requires more memory since more data points have neighbours which are stored in memory. Talking about 2nd SVM implementation it does not differ significantly but slightly reduced CPU time appears from lower number of reported outliers because it skips extra appends to the outlier list.

Table 7: KDD Cup 1999 dataset: Results of algorithms modifications

Algorithm	Peak Memory, KB	CPU Time, s	Recall	Precision
Exact-Storm	11175	98724	92.2%	38.5%
Exact-Storm*	169180	97964	92.8%	82.4%
Exact-Storm**	156592	91281	91.6%	100.0%
Exact-Storm***	163420	96893	91.0%	100.0%
Approx-Storm	98840	99277	92.2%	39.2%
Approx-Storm*	158668	94234	92.8%	82.4%
Approx-Storm**	156076	87200	91.6%	100.0%
Approx-Storm***	153040	111381	91.0%	100.0%
Abstract-C	97972	99292	92.2%	38.5%
Abstract-C*	156080	105032	92.8%	82.4%
Abstract-C**	152872	106792	91.6%	100.0%
Abstract-C***	153228	112317	91.0%	100.0%
MCOD	124640	387	92.2%	38.5%
MCOD*	172204	152	92.2%	82.8%
MCOD**	169896	414	91.6%	100.0%
MCOD***	169828	403	91.0%	100.0%

* implementing weighted distance
** modification of using linear SVM to double check outlierness
*** modification of using non-linear SVM to double check outlierness

KDD Cup 1999 dataset (Table 7) with classical distance-based outlier detection algorithms reached quite good recall which is around 92%, however the precision was not very high (less than 40%). Experimenting with modified algorithms showed that introducing weighted distance increased precision twice compared to classical distance-based outlier detection algorithms. Other modifications - using linear or non-linear SVM as an additio-

nal step for outlier verification increases precision to 100.0% but these modifications as in previous case causes little decrease in a recall.

In this case some of the modifications also caused reduction in CPU time. The main reason behind that is relatively smaller number of outliers reported.

Table 8: Pendigits dataset: Results of algorithms modifications

Algorithm	Peak Memory, KB	CPU Time, s	Recall	Precision
Exact-Storm	79056	4486	78.2%	24.3%
Exact-Storm*	112808	4849	82.1%	35.0%
Exact-Storm**	110552	4521	71.8%	94.1%
Exact-Storm***	110540	4464	76.3%	99.2%
Approx-Storm	89132	4198	66.7%	25.7%
Approx-Storm*	104264	5341	63.5%	33.3%
Approx-Storm**	100664	4329	60.9%	93.1%
Approx-Storm***	110508	4285	64.7%	99.0%
Abstract-C	78548	4354	78.2%	24.3%
Abstract-C*	104320	5501	82.1%	35.0%
Abstract-C**	100608	4716	71.8%	94.1%
Abstract-C***	100960	4691	76.3%	99.2%
MCOD	78084	2819	80.8%	24.6%
MCOD*	114512	2680	82.1%	34.2%
MCOD**	110672	3387	73.1%	91.2%
MCOD***	100468	2912	78.8%	99.2%

* implementing weighted distance
** modification of using linear SVM to double check outlierness
*** modification of using non-linear SVM to double check outlierness

Pendigits dataset (Table 8) with distance-based outlier detection algorithms reach a recall around 66-80% (depending on the algorithm) while the precision is quite small, varies around 25%. Introducing weighted distance had a slight impact on recall and precision. With algorithms Exact-Storm, Abstract-C and MCOD there is a small increase in recall while for Approx-Storm small decrease caused by usage of approximations. Precision on the other hand have a little increase despite what distance-based outlier detection algorithm was used. However this is not appropriate result for precision if so many data points are reported as outliers when in fact they are not. In contrast using linear and non-linear SVM as an additional step for outlier verification increased precision significantly from 25% to 91% and more. However these modifications caused a decrease in recall by 1-7%.

Talking about Pendigits dataset CPU time does not show reduction with applied modifications. This is because of the high dimensionality of the dataset and because outliers are mixed within majority of the data and is difficult to separate.

Table 9: Forest Cover dataset: Results of algorithms modifications

Algorithm	Peak Memory, KB	CPU Time, s	Recall	Precision
Exact-Storm	688508	96952	60.2%	6.8%
Exact-Storm*	655660	655660	98.8%	44.6%
Exact-Storm**	741720	78275	49.3%	98.8%
Exact-Storm***	726648	104817	50.5%	100.0%
Approx-Storm	668472	67491	59.3%	6.8%
Approx-Storm*	671624	117203	98.8%	45.1%
Approx-Storm**	745788	109848	48.5%	98.7%
Approx-Storm***	738500	105253	49.7%	100.0%
Abstract-C	676020	88897	60.0%	6.8%
Abstract-C*	320036	115810	98.8%	44.6%
Abstract-C**	749216	109022	49.3%	98.8%
Abstract-C***	750152	104417	50.5%	100.0%
MCOD	991120	93924	60.4%	6.8%
MCOD*	940240	55287	98.8%	43.4%
MCOD**	940240	99197	49.3%	98.8%
MCOD***	1028692	94773	50.7%	100.0%

* implementing weighted distance
** modification of using linear SVM to double check outlierness
*** modification of using non-linear SVM to double check outlierness

Forest Cover dataset (Table 9) with classical distance-based outlier detection algorithms do not show good results - recall is around 60% while precision is very low, only 6.8%. It means that using distance-based outlier detection algorithms is not very suitable for this king of dataset. But let's see if algorithm modifications can improve outlier detection accuracy. Introducing weighted distance significantly increased recall and precision (recall reached 98.8% and precision 43-45%). Additional step for outlier verification by using linear and non-linear SVM significantly increases precision which reaches even 98-100%. However, such implementation causes recall reduction approximately by 10%. This happens because outliers in the dataset are not clearly separable when we take into consideration all features.

In this case CPU time does not show unambiguous reduction with applied modifications. This is because of the high dimensionality of the dataset and because outliers are mixed within inliers and is difficult to separate.

In general, implementing SVM in distance-based outlier detection algorithms bring some positive effect to the outlier detection accuracy and in some cases even in CPU time reduction. With chosen datasets distance-based outlier detection performs quite differently. Let's discuss in detail all type of situations.

There are datasets to which distance-based outlier detection performs quite well (in

our case - Shuttle dataset). Recall and precision in this case is more than 94%. What happens when we implement SVM - we increase precision while recall stays high.

There are datasets to which distance-based outlier detection gives good recall which means that outliers are identified well but also many other data points are reported as outliers which gives poor precision (in our case - KDD Cup 1999 dataset). Once we train SVM, algorithm learns what usual behaviour of dataset is and this information helps on eliminating incorrectly identified outliers from outlier list. This way precision can be increased.

There are datasets to which distance-based outlier detection gives average recall and very small precision (in our case - Pendigits and Forest Cover datasets). One of the reasons is specifics of the datasets. However, we found out that accuracy still can be improved. Main observations are that introducing weighted-distance improves recall and can increase precision while introducing SVM for double checking the outlierness have a significant increase in the precision but unfortunately recall may drop.

So these are the conclusions presented for different types of datasets. Generalizing the overall results, implementing SVM mostly helps on improving outlier detection accuracy. Introducing weighted distance to distance-based outlier detection algorithms highlights the patterns of labeled outliers in training subset and what feature changes cause data point to be an outlier. Another implementation introduces additional step which performs filtering of data points that have less than k neighbours within distance R in a window. With classical algorithms these data points would be reported as outliers but with this modification we additionally use SVM to check if data point does not fall into the inliers area (these areas are illustrated in Figure 26-29). Main advantage of this implementation is that selected data points are filtered twice and second filtering reduces the number of reported outliers which adds to increasing precision.

2.4. Outliers clustering

Most of the outlier detection algorithms focus on detecting outliers but do not investigate them. In some situations it might happen that outliers repeat and have some frequency which is not easily noticeable especially if we use sliding window approach. By investigating what kind of outlier patterns form during the time we can discover that some outliers repeat and this information can stimulate useful observations. To achieve this, distance-based outlier detection algorithms were supplemented by predicted outliers clustering.

In the algorithms additional parameter n_{cl} appeared which states how often clustering is trained. In our experiments $n_{cl} = 10$. It means that first time K-Means clustering will be trained after first 10 windows are processed. After clusters are formed all new predicted outliers will be assigned to the closest cluster untill the retrain of the clustering is done. In our case retrain will be done every 10 windows.

To find optimal number of clusters k' we will implement Silhouette method which will be recalculated every time we retrain K-Means. It means that number of clusters can change during the time depending on identified outliers.

All the time when outliers are clustered we receive information about the clusters:

- the number of data points belonging to each cluster;
- the average distance between cluster data points and a cluster centroid;
- the mean feature values of data points within a cluster.

Further we will discuss particular examples of clustered outliers and what additional information is received from clustering. In this section for each dataset we chose to show one example of outliers clustering. For predicting outliers we used one of the modified MCOD algorithms which give rather high precision and recall. For discussing the results for each dataset final clustering are presented.

In the following graphs clusters are presented visually. You can find two types of data points: marked as 'x' or 'o'. This specific tagging refers to the real label of visualized data point (in initial dataset the real outliers are labeled). Most of the data points are marked as 'x' as for clustering I chose algorithms which give highest possible precision.

Shuttle dataset has an outlier rate of 7%. By using MCOD with modification of non-linear SVM our detected outlier rate is 6.7%. After clustering detected outliers we receive 2 clusters, they are illustrated in Figure 30. Let's see what additional information about the clusters are received in Figure 31. Cluster '0' (green colour) is a bigger cluster composed of 2379 data points with average distance within the cluster 0.0569. Cluster '1' (red colour) is composed of 926 data points with average distance within the cluster 0.0348. From the variable means by clusters we see that mostly clusters differ by x_0, x_2, x_4, x_7 and

x_8 features. This information indicates what kind of feature differences impact clustering and by comparing it to the rest of the data we can find out what kind of outliers fall into such clusters.

Figure 30: Shuttle: Plotting predicted outliers

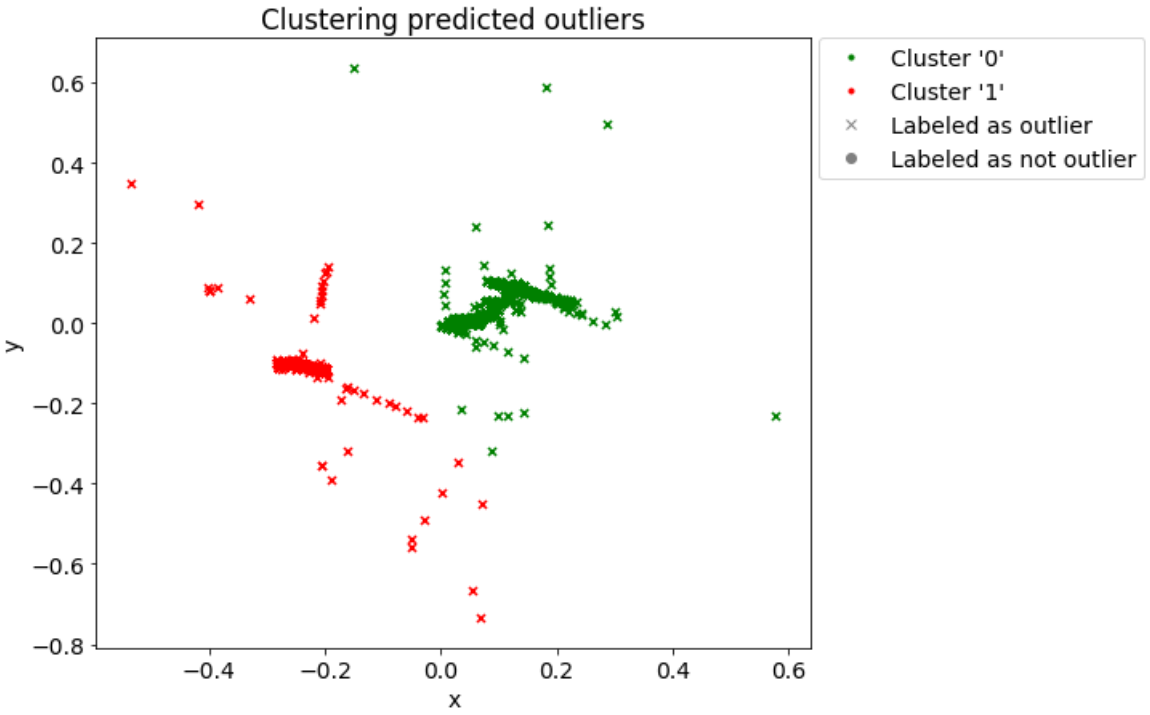


Figure 31: Shuttle: Describing clusters

```
Number of data points in clusters: Counter({0: 2379, 1: 926})
Mean distances within clusters:
0   0.05693812976718062
1   0.03484936522425923
Variable means by clusters:
                0         1         2         3         4         5         6         7         8
cluster
0        0.544194  0.487509  0.497783  0.506802  0.267855  0.637928  0.338748  0.737797  0.737410
1        0.777145  0.492009  0.656528  0.507057  0.413309  0.638287  0.320452  0.622926  0.626982
```

KDD Cup 1999 dataset has an outlier rate of 0.4%. By using MCOD with modification of non-linear SVM our detected outlier rate is 0.36%. After clustering detected outliers we receive 2 clusters, they are illustrated in Figure 32. From the picture we see that cluster '0' is much denser comparing to cluster '1'. Let's see if we can prove this with additional information from Figure 33. Cluster '0' (green colour) is a bigger cluster composed of 2013 data points with average distance within the cluster 0.00067. Cluster '1' (red colour) is composed of 25 data points with average distance within the cluster 0.06729. If we compare these clusters to each other we can state that cluster '0' is much more compacted than cluster '1' and most probably refer to some kind of anomaly which can be named after investigating the feature means.

Figure 32: KDD Cup 1999: Plotting predicted outliers

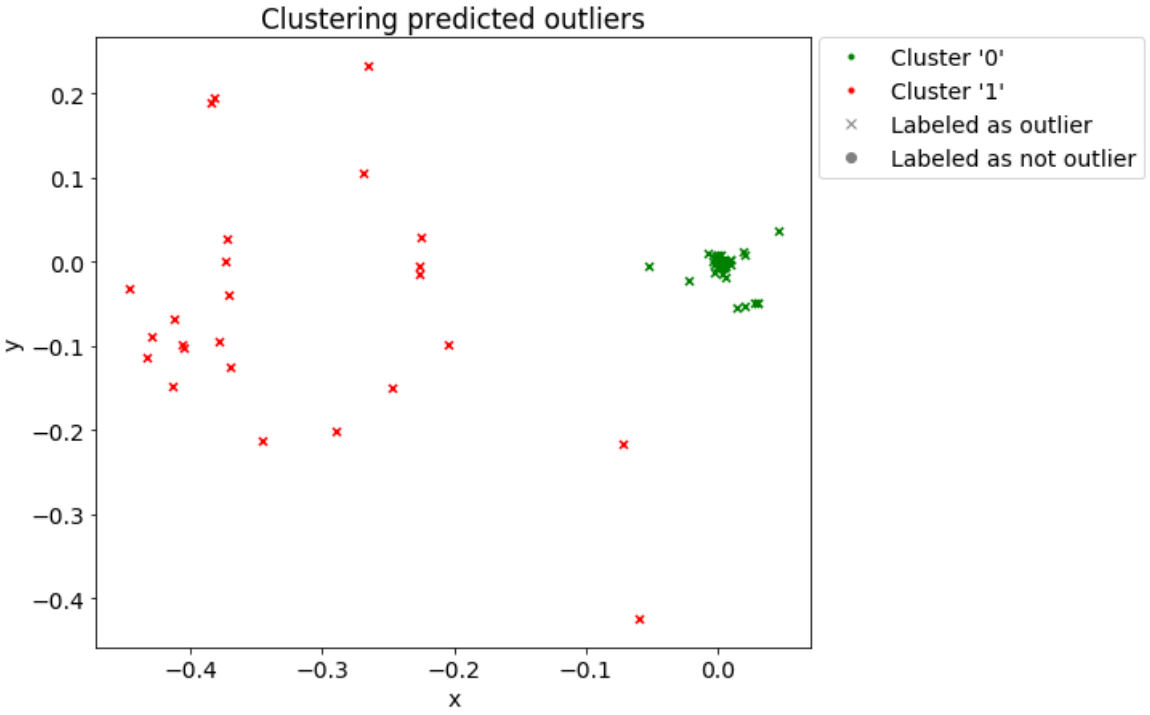


Figure 33: KDD Cup 1999: Describing clusters

```
Number of data points in clusters: Counter({0: 2013, 1: 25})
Mean distances within clusters:
0   0.0006683720971591465
1   0.06729091477740606
Variable means by clusters:
                0         1         2
cluster
0        0.000000  0.999928  0.609377
1        0.365931  1.000000  0.609694
```

Pendigits dataset has an outlier rate of 2.27%. By using MCOD with modification of non-linear SVM our detected outlier rate is 1.64%. After clustering detected outliers we receive 2 clusters, they are illustrated in Figure 34. This example shows that clustering with this dataset is not very useful since all the outliers behave quite similarly and form one cloud. Of course during the time with incoming new data points formed clusters might move and highlight new patterns. Let's see what additional information about the clusters is received in Figure 35. Cluster '0' (green colour) is a bigger cluster composed of 68 data points with average distance within the cluster 0.4175. Cluster '1' (red colour) is composed of 45 data points with average distance within the cluster 0.4312. Variable means by clusters presented in the Figure 35 indicate what are the main differences between the clusters.

Figure 34: Pendigits: Plotting predicted outliers

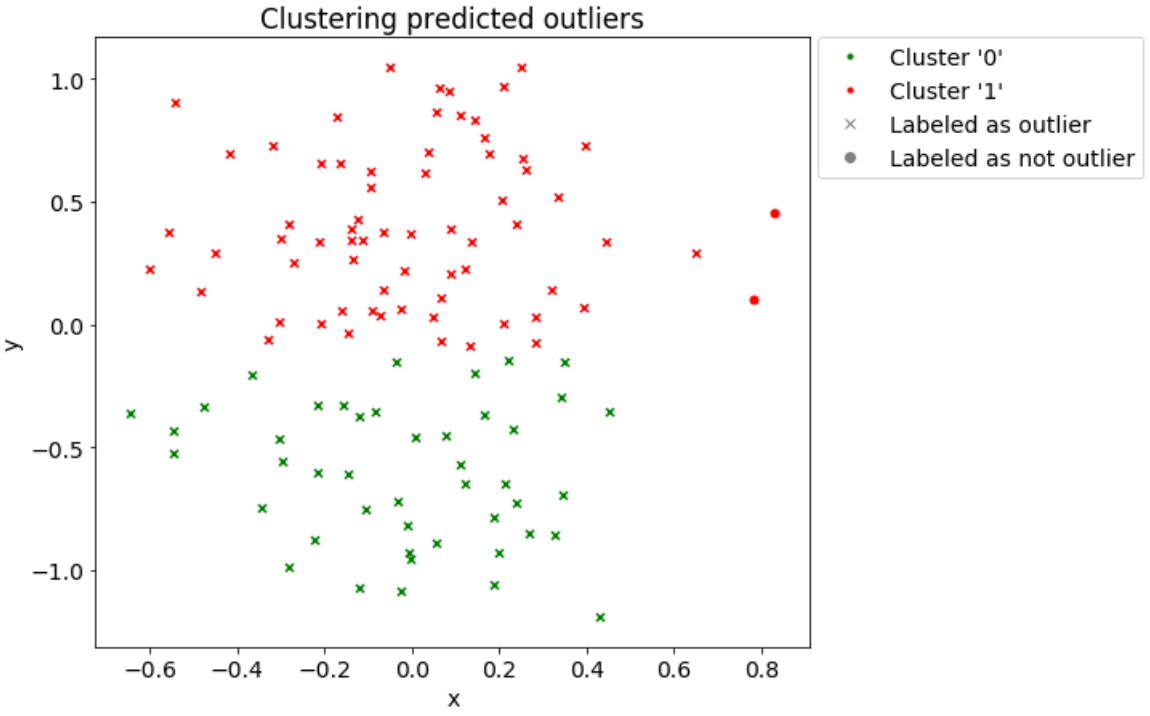


Figure 35: Pendigits: Describing clusters

```
Number of data points in clusters: Counter({1: 68, 0: 45})
Mean distances within clusters:
0   0.4175466021489506
1   0.4311640504454897
Variable means by clusters:
                0         1         2         3         4         5         6         7
cluster
0        0.186863  0.744663  0.077948  0.358251  0.347074  0.011757  0.818169  0.124854
1        0.346225  0.896131  0.075514  0.645255  0.050004  0.208478  0.436343  0.009241

                8         9         10        11        12        13        14        15
cluster
0        0.999248  0.546052  0.766605  0.943894  0.282819  0.946934  0.049576  0.583741
1        0.881204  0.229843  0.987602  0.648921  0.664507  0.957426  0.200582  0.864850
```

Forest Cover dataset has an outlier rate of 0.4%. For outlier clustering we used subset of 50000 data points, in which outlier rate reaches 0.97%. By using MCOD with modification of non-linear SVM our detected outlier rate is 0.67%. After clustering detected outliers we receive 2 clusters, they are illustrated in Figure 36. It is visible that clusters behave a differently, so let's see what additional information about the clusters are received in Figure 37. Cluster '0' (green colour) is a bigger cluster composed of 286 data points with average distance within the cluster 0.27380. Cluster '1' (red colour) is composed of 52 data points with average distance within the cluster 0.23849. Variable means by clusters indicate what are the main differences between clusters. By comparing it to the feature means of all dataset we can define what kind of outliers is in the specified clusters.

Figure 36: Forest Cover: Plotting predicted outliers

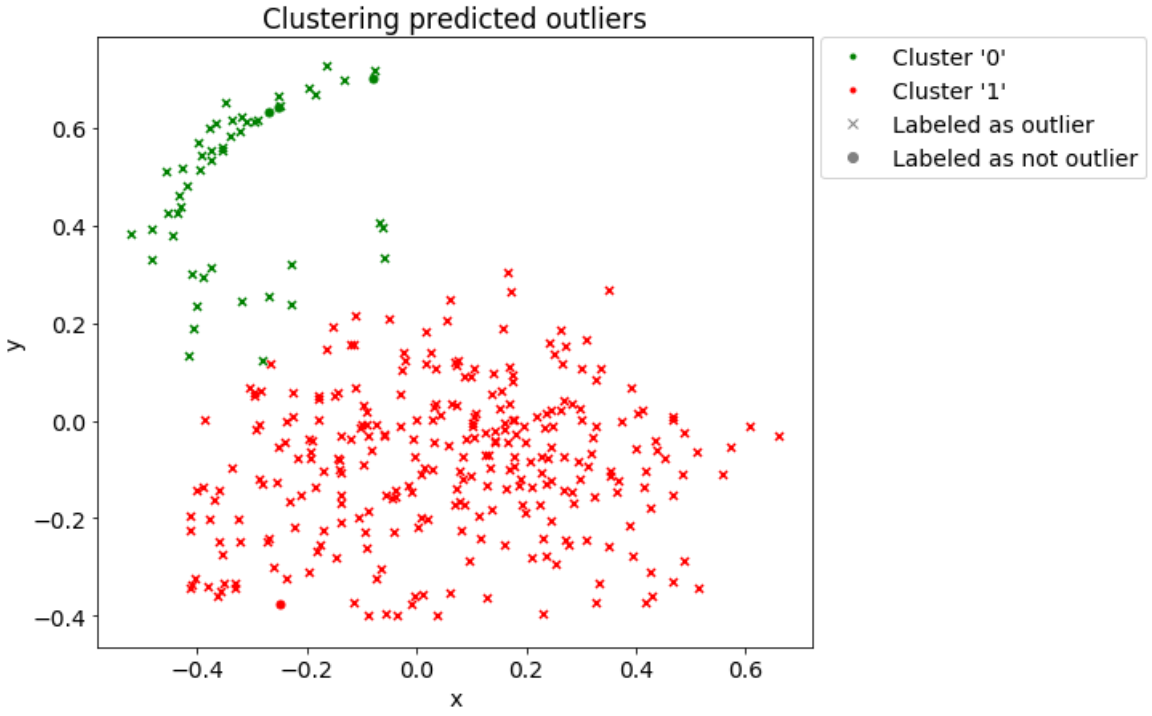


Figure 37: Forest Cover: Describing clusters

```
Number of data points in clusters: Counter({0: 286, 1: 52})
Mean distances within clusters:
0   0.27379941675730707
1   0.23848545157859272
Variable means by clusters:
                0         1         2         3         4         5         6  \
cluster
0        0.142217  0.300272  0.339267  0.072906  0.280565  0.135454  0.948461
1        0.096699  0.803472  0.217075  0.026375  0.236807  0.138601  0.764158

                7         8         9
cluster
0        0.829690  0.335375  0.115047
1        0.910282  0.678755  0.107931
```

To summarize, discussed datasets show how clustering can generalize detected outliers. In some cases, e.g. Pendigits dataset, clustering does not seem to be very useful because all the outliers are very similar. On the contrary with other datasets detected outliers tend to group and that is clearly visible when we visualize the detected outliers.

One of the best examples was received using detected outliers of KDD Cup 1999 dataset which is shown in Figure 32. From the figure we see that one group share were similar characteristics (cluster '0'). Once this trend is spotted using summary of the cluster we can identify the origin of such outliers and use this information in further analysis.

Even though in all of these examples optimal number of clusters was $k' = 2$, in real-world more clusters are expected to appear. Introducing outliers clustering helps generalizing outlier tendencies and can ease understanding newly predicted outliers.

2.5. Conclusions of experimental results

To generalize, Section 2 was dedicated to discussing the results when described distance-based algorithms and suggested modifications were used for outlier detection with stream data.

We used 4 datasets and presented the results for each of them separately. Datasets are quite different and that makes analysis more interesting since we could discuss how specific dataset reacts to distance-based outlier detection approach.

In one of the sections we ran distance-based outlier detection algorithms with all the datasets while varying algorithm parameters: neighbours count k, distance R, window size W and slide size S. Let's discuss if choice of these parameters have influence on outlier detection algorithms performance.

Parameter k in distance-based outlier detection defines the neighbour threshold that is required for data point so it would be assigned to the inliers. Results for all of datasets with varying parameter k were discussed separately and some common conclusions were made. If we fix other parameters (R, W and S) and increase the value k we see that outlier detection accuracy is affected in a way that bigger k increases recall but decreases precision.

Parameter R defines the distance threshold which is used to define the area around data point in which neighbours search is performed. Parameter R was chosen for each dataset separately according to the specifics of the dataset. If we fix other parameters (k, W and S) and vary R we observe that choice of R mostly affects outlier detection accuracy. Common observations are that once we increase the distance R outlier detection recall is decreasing and precision increasing meaning that smaller number of data points are reported as outliers.

Parameter W defines how many data points are included into a window in which outlier detection is processed. If we fix all other parameters (k, R and S) and vary window size we get constantly decreasing recall and precision. To make it more interesting we decided to fix two parameters (R and S) and with increasing W respectively increment neighbour count threshold k. The hypothesis were raised that changed W have little impact on the outlier detection accuracy if we adjust neighbour count threshold k respectively. After performing the experiments we have proven the hypothesis and added that the only effect that increased W have is the constantly increasing processing time.

Parameter S defines the step by how many data points the window is slided on each iteration. For all the datasets we chose the same S values to investigate while the rest of the parameters (k, R and W) were fixed. What we observed is that chosen S mostly affects algorithm processing time and has no significant impact on accuracy. Smaller S means that more iterations for outlier detection are done with including newest data by smaller portions.

After these experiments were done we were able to identify best possible parameters which help to achieve best results by using distance-based outlier detection algorithms for stream data. Since all of the applied algorithms are from the same distance-based algorithms class observed outlier detection accuracy results were quite similar. More interesting was seeing that different datasets also differently reacted to distance-based approach. Table 10 shows the summary of the observations where for each dataset average results obtained by using different algorithms are presented.

Table 10: Results of applying distance-based outlier detection algorithms on 4 datasets

Dataset	k	R	Peak Memory, KB	CPU Time, s	Recall	Precision	Outlier rate*
Shuttle	55	0.25	90969	93989	95.0%	96.0%	7.08% / 7.00%
KDD Cup 1999	5	0.35	83157	74420	92.2%	38.7%	0.80% / 0.40%
Pendigits	15	1.1	81205	3964	76.0%	24.7%	7.00% / 2.27%
Forest Cover	10	0.4	774563	90696	60.0%	6.8%	8.61% / 0.90%

* In this column outlier rate of predicted outliers versus real dataset outlier rate are presented

As seen from the table distance-based outlier detection works very good with Shuttle dataset having recall of 95% and precision 96%. Another dataset called KDD Cup 1999 using distance-based algorithms get relatively good recall while precision does not reach 40%. Two more datasets considered in the experimental part are Pendigits and Forest Cover. These datasets are the examples where distance-based outlier detection algorithms are not working very well. Obtained recall in this case is around 76% and 60% while precision is very low having the values of 24.7% and 6.8% respectively.

Speaking about algorithms performance, in respect to outlier detection accuracy all of them performed very similar and this is because the same distance-based approach is used. Thinking about algorithms processing time, mostly stands out MCOD: in cases where outliers can be separated more easily MCOD is significantly faster but on opposite situations or with wrongly chosen parameters its processing time can vastly exceed a usual CPU time. In contrary Abstract-C shows most stable results among analysed algorithms. It does not show better accuracy than others or does not significantly stands out with respect to CPU time but it is most stable among analysed algorithms.

After applying distance-based outlier detection algorithms we saw that in some cases these algorithms are not giving good accuracy. Once we investigated the datasets we observed that the reason behind this is that all of the dataset features are treated equally and because of that it becomes hard to separate the outliers within multidimensional data. So we considered a training subset and ran SVM that learnt the behaviour of the labeled outliers in given subset. This information was implemented and used with distance-based outlier detection algorithms.

Changing the classical distance to weighted distance where weights are assigned using trained linear SVM can improve outlier detection accuracy. After comparing the results

of modified algorithms with classical distance-based outlier detection algorithms we saw that introduced weights suggest what features are most relevant for deciding data point outlierness. This modification helps increasing both: outlier detection recall and precision.

Another implementation that was suggested is using trained SVM to filter outliers reported by classical distance-based outlier detection algorithms. The implementation mostly helps dealing with low precision.

Table 11: Results of classical and modified distance-based outlier detection algorithms

Dataset	Peak Memory, KB	CPU Time, s	Recall	Precision	Outlier rate*
Introducing weighted distance					
Shuttle	125760	76662	95.0%	98.0%	6.89% / 7.00%
KDD Cup	164033	74346	92.7%	82.5%	0.38% / 0.40%
Pendigits	108976	4593	77.5%	34.4%	5.44% / 2.27%
Forest Cover	646890	235990	98.8%	44.4%	2.22% / 0.90%
Introducing additional step of outlier verification (linear SVM)					
Shuttle	117558	92301	94.2%	99.9%	6.74% / 7.00%
KDD Cup	158859	71422	91.6%	100.0%	0.31% / 0.40%
Pendigits	105624	4238	69.4%	93.1%	1.76% / 2.27%
Forest Cover	794241	99086	49.0%	98.8%	0.49% / 0.90%
Introducing additional step of outlier verification (non-linear SVM)					
Shuttle	115428	88390	94.0%	100.0%	6.73% / 7.00%
KDD Cup	159879	80249	91.0%	100.0%	0.31% / 0.40%
Pendigits	105619	4088	74.0%	99.2%	1.80% / 2.27%
Forest Cover	810998	102315	50.4%	100.0%	0.49% / 0.90%

* Outlier rate of predicted outliers versus real dataset outlier rate is presented

Table 11 above summarizes the average results obtained by using distance-based outlier detection algorithms with suggested implementations. First improvement which consists of using weighted-distance improved both: recall and precision. The greatest results were achieved for KDD Cup 1999 dataset where precision was increased from 38.7% to 82.5% and for Forest Cover dataset where recall was increased from 60.0% to 98.8%. Second implementation of SVM that consists of additional filtering of outliers mostly helps improving precision. By implementing linear SVM with all datasets we achieved precision higher than 93.0%. By using non-linear SVM we achieve even higher precision which reaches 99.0% or more. However disadvantage of this improvement is that additional filtering can reduce recall up to 10.0%

Other area we focused on was outliers clustering which promotes additional information about predicted outliers. We introduced additional clustering parameter which defines how often clustering is retrained. In between the training, new outliers are assigned to one of the existing clusters. By looking into characteristics of the clusters we can tell what kind of anomaly patterns form and what causes these outliers to appear.

Conclusions

Outlier detection is well known problem in mathematics which seeks to identify observations that significantly deviates from the common behaviour in the dataset. Nowadays when real-time decisions become more and more valuable one of the interesting areas where outlier detection can be applied is outlier detection in stream data.

There are many different outlier detection algorithms proposed which work with stream data. Usually they are divided into groups based on the specifics of the algorithm. In this paper we focused on distance-based outlier detection algorithms. They are proven to be effective, widely applicable, highly scalable, it is easy to understand why certain decision about data point outlierness is made and no information about the data distribution is required which is very difficult to define for multidimensional data. In theoretical part of the paper we presented 4 distance-based outlier detection algorithms: Exact-Storm, Approx-Storm, Abstract-C and MCOD. Later on they were used in experimental part.

One of the tasks in the experimental part was to apply mentioned algorithms to the chosen real-world examples, investigate how outlier detection works with different datasets and analyse the effect of parameters to the overall outlier detection result.

These are the main observations how change of parameters influence outlier detection algorithms:

- With increasing neighbour count k the recall increase but precision decrease.
- With increasing distance R the recall decrease but precision increase.
- Increased window size W constantly increase algorithm processing time.
- Decreased slide size S increases algorithm processing time.

If we talk about algorithms performance, in respect to outlier detection accuracy all the algorithms performed very similar. Thinking about algorithms processing time, mostly stands out MCOD: with well selected parameters MCOD is significantly faster but on opposite situations its processing time can vastly increase. In contrary Abstract-C shows most stable results among analysed algorithms.

Despite the fact that these algorithms are defined as effective for outlier detection it does not consider any additional information about the dataset which can help to improve identification of outliers. By investigating the specifics of the datasets we saw that in some cases labeled outliers are recognized by the change of specific features while the rest of the features behave same as common inliers features.

One of the suggested implementations for improving performance of distance-based outlier detection algorithms is related to usage of SVM. Using training sample SVM learns the behaviour of the data and labeled outliers. Afterwards we use SVM with distance-based outlier detection by applying one of the modifications: using linear SVM and trained

hyperplane to learn the importance of the features, using trained linear SVM for reported outliers filtering or using trained non-linear SVM for reported outliers filtering.

First suggested improvement is using linear SVM coefficients for weighted distance. Fortunately this modification did not have any negative effect on outlier detection accuracy - for all investigated datasets recall and precision increased or at least stayed the same. Best achieved results were: increased recall from 60.0% to 98.8% and increased precision from 38.7% to 82.5% (for different datasets).

Another improvement which consists of adding additional step in the stage of outliers reporting considered two SVM training cases: using linear and non-linear SVM. In general this modification helps dealing with low precision. For a dataset which with classical distance-based algorithms had very low precision (6.8%) we managed to increase it to 98.8% by using linear SVM and to 100.0% by using non-linear SVM.

One more add-on to the distance-based outlier detection algorithms is predicted outliers clustering. Usually outlier detection algorithms simply return the list of outliers which says little about the reported data points. However, in some cases especially when we are working with stream data similar outliers may repeat with some frequency. This is why outliers clustering were introduced. What we received from the clusters is that similar outliers gathers together and we can monitor if some significant anomaly patterns forms. In one of the investigated datasets (KDD Cup 1999) two clusters were formed one of which was very dense. This observation indicates that those outliers are very similar. By investigating the cluster we can find out the origin of these outliers.

Generalizing the work done in the thesis, we investigated that distance-based outlier detection algorithms work well with the datasets where outliers are clearly separable. However, in the multidimensional datasets where outlierness is observed by deviation of few specific features, distance-based outlier detection algorithms do not catch such outliers. We tested suggested modification of implementing SVM which helps learning the behaviour of outliers and inliers, and after comparing the results we found out that suggested modifications can add a value to outlier detection accuracy.

Since in the experimental section we used only the basic SVM functions (linear SVM and non-linear SVM based on RBF function) further work could focus on investigating which SVM finds best separation for outlier detection and combined with distance-based algorithms performs best accuracy. Another possible area to develop in the future is related to outliers clustering. Once some anomaly patters are recognized which are defined as possible rare events, our SVM can be retrained by initializing that this kind of data points actually are not outliers but observations that occurs rarely. This way retrained SVM would learn some additional information about the dataset and in long run outlier detection hopefully would be improved.

References

[1] Ramesh Kumar and Aljinu Khadar. A survey on outlier detection techniques in dynamic data stream. *International Journal of Latest Engineering and Management Research (IJLEMR)*, 2(8):23–30, 2017.

[2] Cheong Hee Park. Outlier and anomaly pattern detection on data streams. *The Journal of Supercomputing*, 75:6118–6128, 2019.

[3] Manish Gupta, Jing Gao, Charu C. Aggarwal, and Jiawei Han. Outlier detection for temporal data: A survey. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 26(9):2250–2267, 2014.

[4] Luan Tran, Liyue Fan, and Cyrus Shahabi. Distance-based outlier detection in data streams. *Proceedings of the VLDB Endowment*, 9(12):1089–1100, 2016.

[5] Susik Yoon, Jae-Gil Lee, and Byung Suk Lee. Nets: Extremely fast outlier detection from a data stream via set-based processing. *Proceedings of the VLDB Endowment*, 12(11):1303–1315, 2019.

[6] Maria Kontaki, Anastasios Gounaris, Apostolos N. Papadopoulos, Kostas Tsichlas, and Yannis Manolopoulos. Continuous monitoring of distance-based outliers over data streams. *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE), Hannover, Germany*, 2011.

[7] Fabrizio Angiulli and Fabio Fassetti. Detecting distance-based outliers in streams of data. *CIKM'07, Lisboa, Portugal*, 2007.

[8] Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. Neighbor-based pattern detection for windows over streaming data. *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 529–540, 2009.

[9] Dragoljub Pokrajac, Aleksandar Lazarevic, and Longin Jan Latecki. Incremental local outlier detection for data streams. *IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, 2007.

[10] Mahsa Saleh, Christopher Leckie, James C. Bezdek, Tharshan Vaithianathan, and Xuyun Zhang. Fast memory e cient local outlier detection in data streams. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 28(12):3246–3260, 2016.

[11] Swee Chuan Tan, Kai Ming Ting, and Tony Fei Liu. Mining streaming and temporal data: from representation to knowledge. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*.

[12] Aaron Tuor, Samuel Kaplan, Brian Hutchinson, Nicole Nichols, and Sean Robinson. Deep learning for unsupervised insider threat detection in structured cybersecurity data streams. *Proceedings of AI for Cyber Security Workshop at AAAI*, 2017.

[13] Mr. Kiran V. Markad, Mr. Kiran M. Moholkar, and Mr. Sopan N. Abdal. Unsupervised distance based detection of outliers by using anti-hubs. *International Research Journal of Engineering and Technology (IRJET)*, 04(01):1350–1355, 2017.

[14] Korn Poonsirivong and Chanintorn Jittawiriyanukoon. A rapid anomaly detection technique for big data curation. *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–6, 2017.

[15] Himani Bhavsar and Mahesh H. Panchal. A review on support vector machine for data classification. *International Journal of Advanced Research in Computer Engineering and Technology (IJARCET)*, 1(10):185–189, 2012.

[16] Yin-Wen Chang and Chih-Jen Lin. Feature ranking using linear svm. *JMLR: Workshop and Conference Proceedings*, (3):53–64, 2008.

[17] Taegong Kim and Cheong Hee Park. Anomaly pattern detection for streaming data. *Expert Systems With Applications*, 149:113252, 2020.

[18] Trupti M. Kodinariya and Dr. Prashant R. Makwana. Review on determining number of cluster in k-means clustering. *International Journal of Advance Research in Computer Science and Management Studies*, 1(6):90–95, 2013.

[19] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. *Appearing in Proceedings of the 23rd International Conference on Machine Learning, Pittsburgh, PA*, 2006.

[20] UCI KDD Archive. http://kdd.ics.uci.edu.

[21] Outlier Detection DataSets (ODDS) library. http://odds.cs.stonybrook.edu/.

Appendix Nr. 1.

Additional tables and graphs for describing datasets

	X0	X1	X2
count	565287.00	565287.00	565287.00
mean	-2.27	5.54	7.48
std	0.46	0.28	1.32
min	-2.30	-2.30	-2.30
25%	-2.30	5.38	6.49
50%	-2.30	5.52	7.41
75%	-2.30	5.72	8.35
max	8.10	10.91	16.28

Figure 38: KDD Cup 1999: Describing data labeled as not outliers

	X0	X1	X2
count	2211.00	2211.00	2211.00
mean	-2.22	10.86	8.99
std	0.57	0.57	0.45
min	-2.30	-2.30	-2.30
25%	-2.30	10.91	9.03
50%	-2.30	10.91	9.03
75%	-2.30	10.91	9.03
max	2.65	10.91	9.03

Figure 39: KDD Cup 1999: Describing data labeled as outliers

Figure 40: KDD Cup 1999: Dataset visualization (visualized first 10000 data points). Red lines are data points that are labeled as outliers.

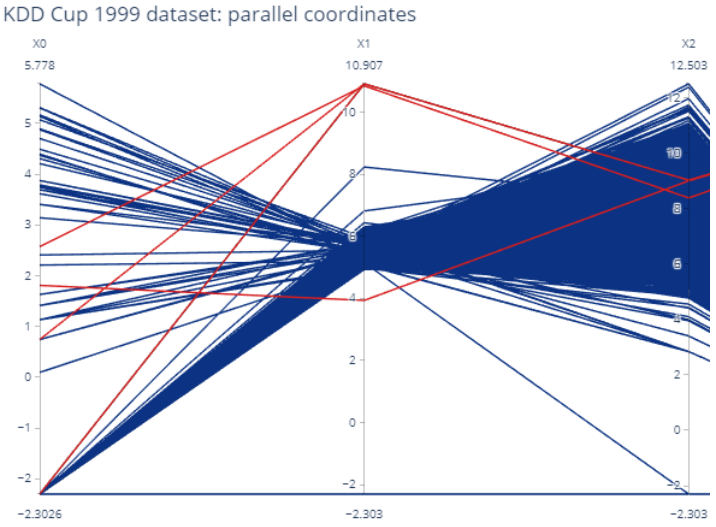


Figure 41: Shuttle: Describing data labeled as not outliers

	X0	X1	X2	X3	X4	X5	X6	X7	X8
count	45586.00	45586.00	45586.00	45586.00	45586.00	45586.00	45586.00	45586.00	45586.00
mean	43.99	-0.50	84.73	0.31	39.27	2.25	40.75	45.45	4.89
std	6.48	31.96	8.67	37.44	14.06	170.63	9.84	15.49	9.77
min	27.00	-4048.00	21.00	-3939.00	-188.00	-12809.00	-16.00	-258.00	-298.00
25%	37.00	0.00	79.00	0.00	36.00	-4.00	35.00	35.00	0.00
50%	43.00	0.00	82.00	0.00	42.00	0.00	40.00	41.00	2.00
75%	49.00	0.00	87.00	0.00	46.00	6.00	43.00	51.00	2.00
max	68.00	318.00	149.00	3830.00	336.00	15164.00	105.00	265.00	226.00

Figure 42: Shuttle: Describing data labeled as outliers

	X0	X1	X2	X3	X4	X5	X6	X7	X8
count	3511.00	3511.00	3511.00	3511.00	3511.00	3511.00	3511.00	3511.00	3511.00
mean	85.11	5.61	90.24	-1.03	5.80	1.05	5.11	85.16	79.95
std	14.27	294.93	9.86	39.34	44.35	537.23	9.56	36.62	37.68
min	37.00	-4821.00	29.00	-2044.00	-188.00	-26739.00	-48.00	-353.00	-356.00
25%	79.00	0.00	84.00	0.00	-36.00	-2.00	2.00	38.00	36.00
50%	82.00	0.00	86.00	0.00	-4.00	0.00	4.00	91.00	88.00
75%	101.00	1.00	102.00	0.00	70.00	5.00	4.00	119.00	114.00
max	126.00	5075.00	112.00	365.00	436.00	8098.00	75.00	270.00	266.00

Figure 43: Shuttle: Dataset visualization (visualized first 10000 data points). Red lines are data points that are labeled as outliers.

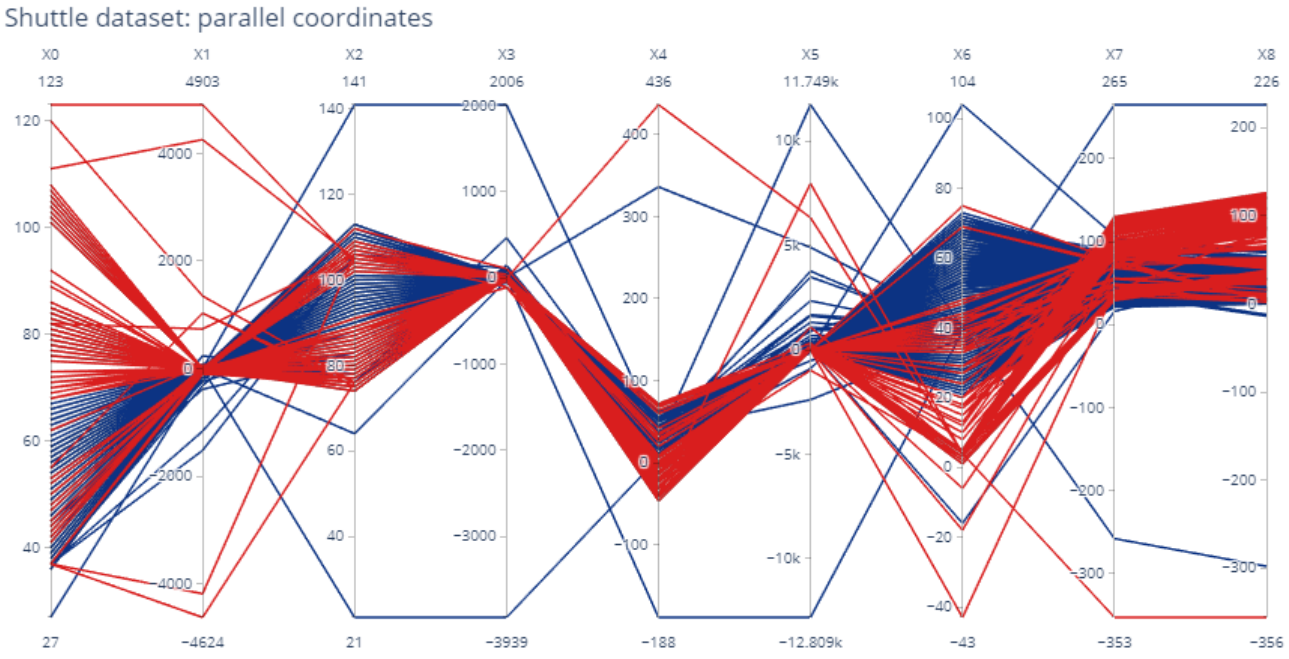


Figure 44: Pendigits: Describing data labeled as not outliers

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15
count	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00	6714.00
mean	0.38	0.84	0.44	0.86	0.55	0.71	0.51	0.49	0.54	0.34	0.58	0.32	0.55	0.30	0.50	0.23
std	0.34	0.17	0.25	0.17	0.34	0.22	0.31	0.29	0.33	0.28	0.38	0.25	0.22	0.29	0.42	0.33
min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
25%	0.01	0.76	0.26	0.75	0.24	0.56	0.27	0.30	0.26	0.06	0.20	0.10	0.42	0.05	0.00	0.00
50%	0.31	0.89	0.43	0.94	0.61	0.74	0.54	0.47	0.55	0.34	0.70	0.27	0.53	0.25	0.48	0.04
75%	0.63	1.00	0.60	1.00	0.84	0.88	0.75	0.68	0.84	0.56	0.95	0.51	0.68	0.38	1.00	0.36
max	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Figure 45: Pendigits: Describing data labeled as outliers

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15
count	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00	156.00
mean	0.33	0.84	0.11	0.56	0.17	0.16	0.56	0.07	0.89	0.35	0.88	0.74	0.54	0.91	0.18	0.72
std	0.22	0.17	0.12	0.24	0.20	0.17	0.27	0.12	0.14	0.24	0.17	0.22	0.26	0.12	0.18	0.24
min	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.32	0.00	0.00	0.15	0.00	0.46	0.00	0.05
25%	0.17	0.74	0.00	0.40	0.00	0.00	0.36	0.00	0.81	0.19	0.77	0.60	0.34	0.84	0.01	0.57
50%	0.30	0.89	0.07	0.60	0.10	0.11	0.58	0.00	0.99	0.30	0.98	0.73	0.56	0.97	0.14	0.75
75%	0.46	0.97	0.17	0.71	0.31	0.28	0.77	0.12	1.00	0.54	1.00	0.95	0.75	1.00	0.28	0.91
max	0.96	1.00	0.63	1.00	0.80	0.67	1.00	0.59	1.00	1.00	1.00	1.00	1.00	1.00	0.77	1.00

Figure 46: Pendigits: Dataset visualization (visualized first 5000 data points). Red lines are data points that are labeled as outliers.

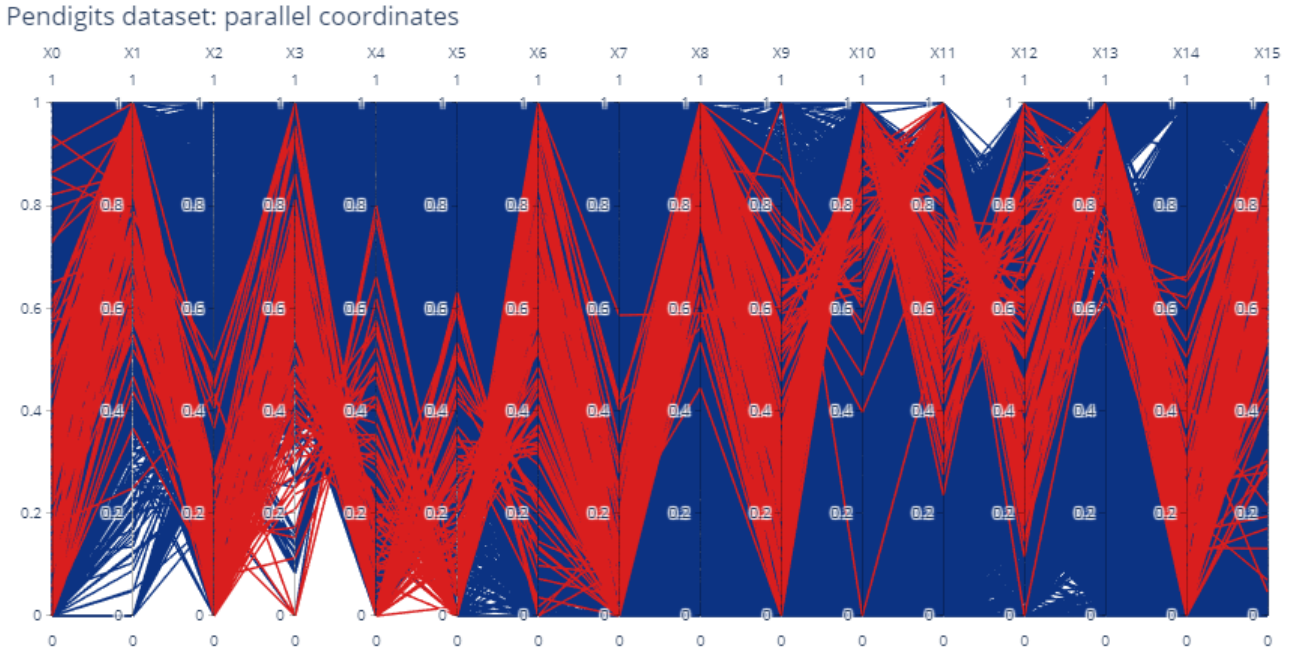


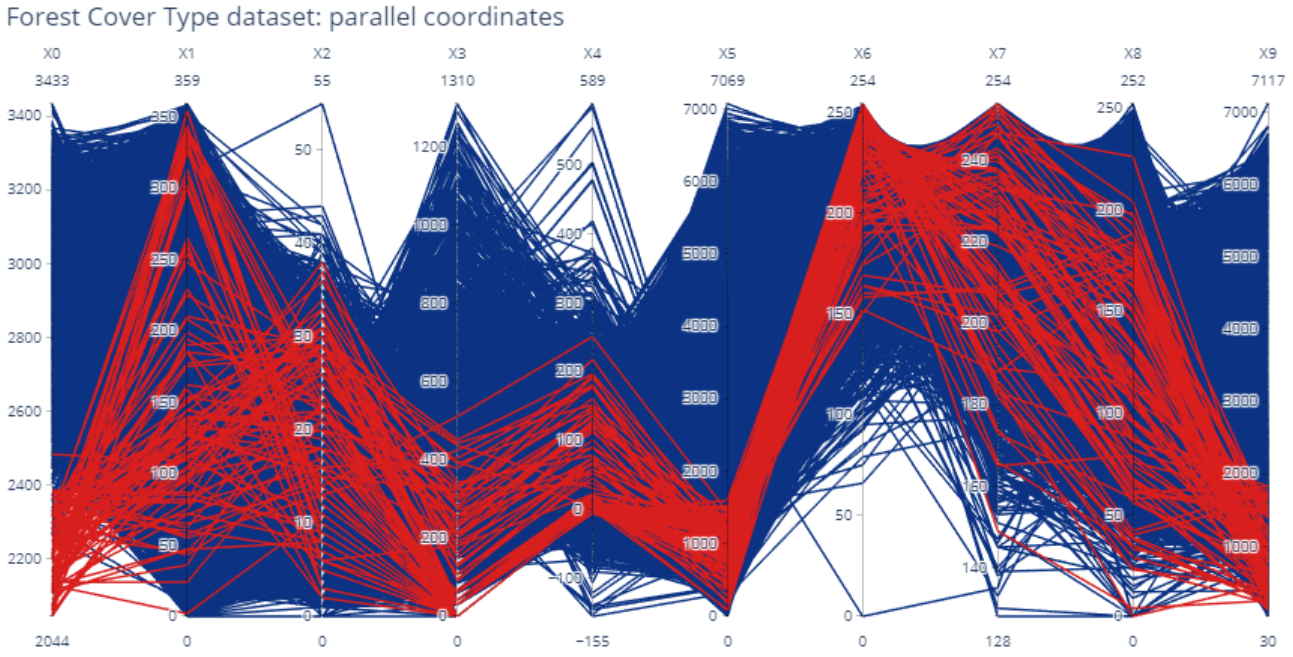
Figure 47: Forest Cover: Describing data labeled as not outliers

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9
count	283301.00	283301.00	283301.00	283301.00	283301.00	283301.00	283301.00	283301.00	283301.00	283301.00
mean	2920.94	152.06	13.55	279.92	45.88	2429.53	213.84	225.33	142.98	2168.15
std	186.58	107.66	7.10	210.35	57.49	1618.72	24.92	18.51	36.22	1424.32
min	2142.00	0.00	0.00	0.00	-173.00	0.00	0.00	0.00	0.00	0.00
25%	2794.00	60.00	8.00	120.00	8.00	1138.00	201.00	215.00	120.00	1173.00
50%	2935.00	127.00	13.00	240.00	30.00	2039.00	219.00	227.00	142.00	1846.00
75%	3042.00	241.00	18.00	391.00	66.00	3408.00	232.00	239.00	167.00	2656.00
max	3433.00	360.00	66.00	1397.00	601.00	7117.00	254.00	254.00	254.00	7173.00

Figure 48: Forest Cover: Describing data labeled as outliers

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9
count	2747.00	2747.00	2747.00	2747.00	2747.00	2747.00	2747.00	2747.00	2747.00	2747.00
mean	2223.94	137.14	18.53	106.93	41.19	914.20	228.35	217.00	111.39	859.12
std	102.52	87.00	9.35	139.74	59.05	366.29	24.14	20.92	49.26	480.86
min	1988.00	0.00	0.00	0.00	-25.00	67.00	127.00	137.00	0.00	0.00
25%	2142.00	83.50	11.00	0.00	0.00	624.00	215.00	204.00	74.00	466.00
50%	2231.00	119.00	19.00	30.00	6.00	949.00	235.00	220.00	113.00	806.00
75%	2304.00	159.00	26.00	192.00	72.00	1218.00	249.00	231.00	149.00	1248.00
max	2526.00	359.00	46.00	551.00	270.00	1702.00	254.00	254.00	232.00	1921.00

Figure 49: Forest Cover: Dataset visualization (visualized first 10000 data points). Red lines are data points that are labeled as outliers.



Appendix Nr. 2.

Additional graphs from Section 2.3

Figure 50: KDD Cup 1999: coefficients of linear SVM

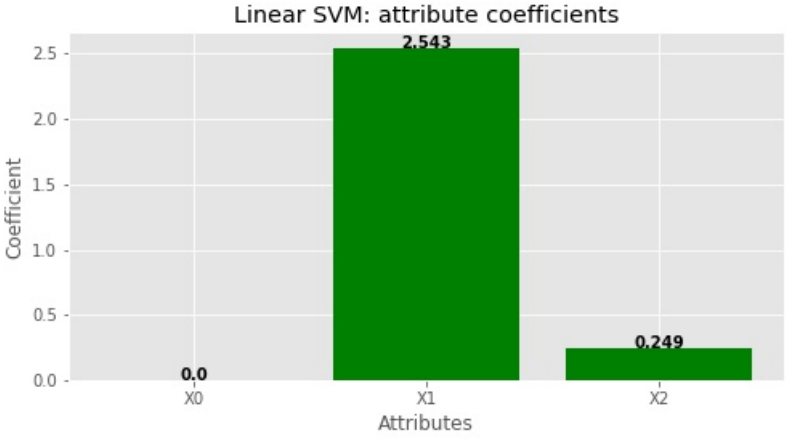


Figure 51: Pendigits: coefficients of linear SVM

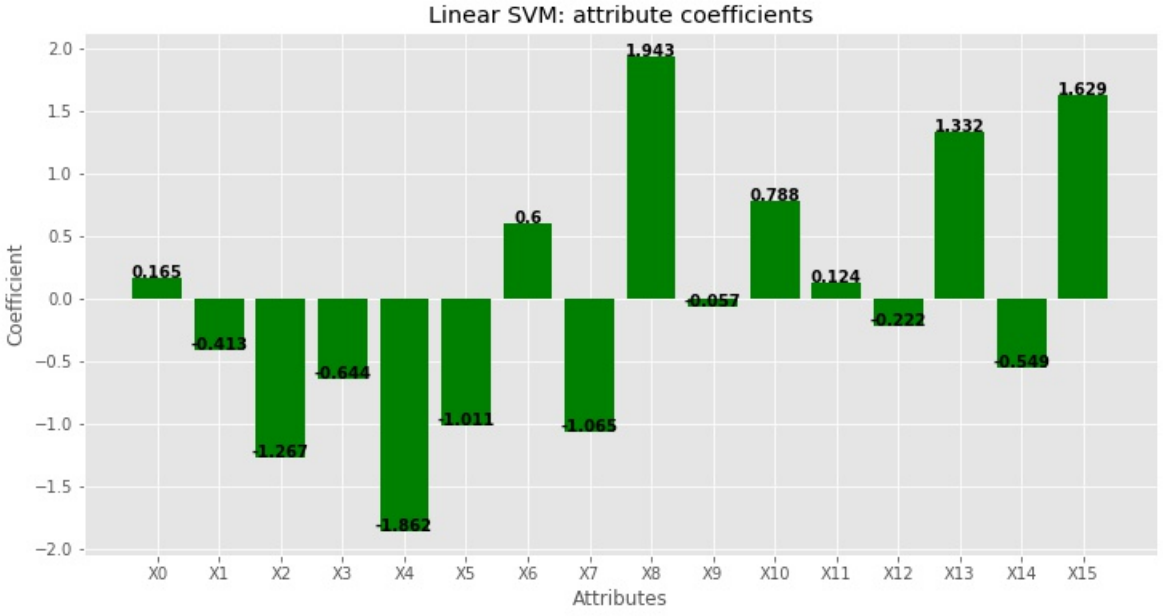
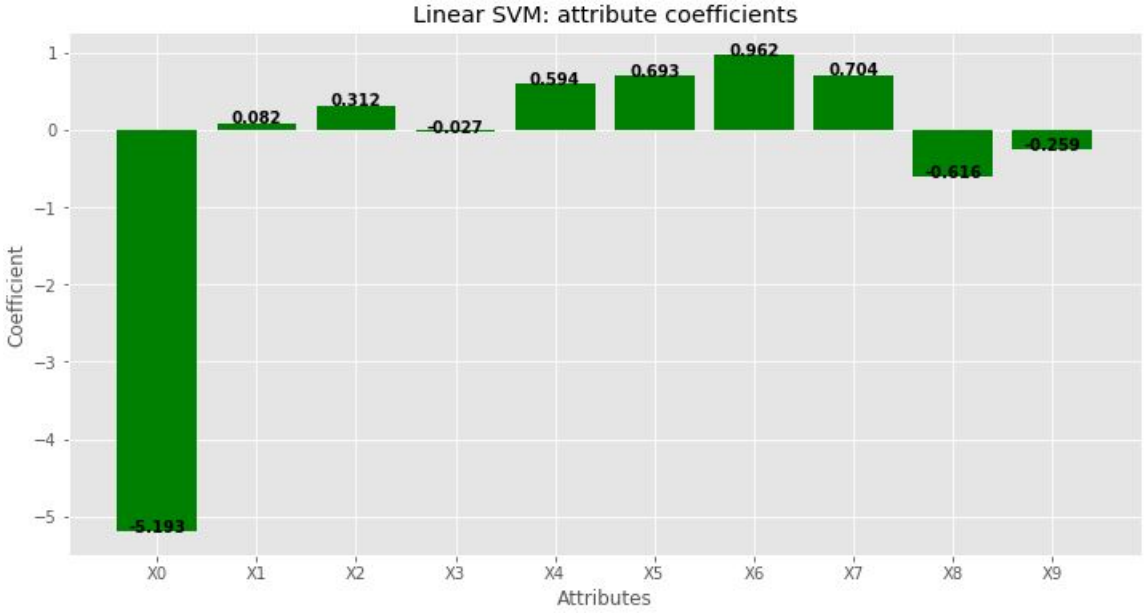


Figure 52: Forest Cover: coefficients of linear SVM



Appendix Nr. 3.

Code used for algorithms

Exact-Storm

```
import timeit
import sys
import psutil
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from collections import Counter, defaultdict
import matplotlib.lines as mlines

# # Defining algorithm functions
def isSameSlide(o1, o2):
    return math.floor(o1.arrivalTime/slide)==math.floor(
            o2.arrivalTime/slide)

def detectOutlier(data,currentTime,W,slide):
    global timeForProcessingWindow
    global dataList
    global outliers
    startTime = timeit.default_timer()
    # remove expired data from dataList
    if (slide != W):
        if len(dataList) != 0:
            d_lim = -1
            for i in range(0,len(dataList),1):
                d = dataList[i]
                if (d <= currentTime - W):
                    d_lim = i
            for j in range(d_lim,-1,-1):
                dataList.pop(j)
    else:
```

```
        dataList = []

    # process new slide
    for dac in data.arrivalTime: #all window
        # do range query for ob
        query = pd.DataFrame(columns = ['col1', 'col2'])
        for nei in data.arrivalTime:
            if dac != nei:
                if (d_int(dac,nei) <= R):
                    query = query.append({'col1' : dac, 'col2' : nei},
                                         ignore_index = True)
        queryResult = []
        for ri in range(0,len(query),1):
            queryResult.append(query.iloc[ri].col2)
        queryResult = sorted(queryResult)
        for dod in queryResult:
            dod = int(dod)
            if (dod > currentTime - W):
                if isSameSlide(data_all.iloc[dod], data_all.iloc[dac]):
                    data_all.at[dac,'count_after'] += 1
                else:
                    if len(data_all.at[dac,'nn_before']) < k:
                        data_all.iloc[dac].nn_before.insert(0,dod)
                            # store object into dataList
        if dac not in dataList: dataList.append(dac)

    # (!) further two cases are presented. Only one should be uncommented
    # (c1) add result to outliers (for classical alg and 1st modification)
    for d in dataList:
        # Count preceeding neighbours
        pre = 0
        for nn_before in (data_all.at[d,'nn_before']):
            if (nn_before > currentTime - W):
                pre = pre + 1
        if (pre+data_all.at[d,'count_after'] < k) and (d not in outliers):
            outliers.append(d)
            data_ev.at[d,'Y_pred'] = 1
#     # (c2) add result to outliers (for 2-3 modifications)
#     for d in dataList:
#         # Count preceeding neighbours
#         pre = 0
```

```
#         for nn_before in (data_all.at[d,'nn_before']):
#             if (nn_before > currentTime - W):
#                 pre = pre + 1
#         if (pre+data_all.at[d,'count_after']<k) and (d not in outliers):
#             a = data_all.iloc[x,:le]
#             a = a.values.reshape(1,-1)
#             svm_pred = classifier.predict(a)
#             if (svm_pred[0] != 0):
#                 outliers.append(d)
#                 data_ev.at[d,'Y_pred'] = 1

    timeForProcessingWindow += timeit.default_timer() - startTime
    return outliers

# # For modifications
# (!) Further extra step for 3 modifications.Only one should be uncommented
data_all = data_all[:1000]
le = len(data_all.columns) - 1
Y = data_all[le]
Y = Y.values
data_all = data_all.drop([le], axis = 1)
# 1. Learning weights for weighted distance
classifier = SVC(C=1.0, kernel='linear', random_state=241)
classifier.fit(data_all, Y)
coef = classifier.coef_
coef = pd.DataFrame(data=coef)
coef = abs(coef)
def d_int(x, y):
    return distance.euclidean(coef.iloc[0,:]*data_all.iloc[x],
                              coef.iloc[0,:]*data_all.iloc[y])
# 2. Training linear SVM
# classifier = SVC(C=1.0, kernel='linear', random_state=241)
# classifier.fit(data_all, Y)
# def d_int(x, y):
#     return distance.euclidean(data_all.iloc[x], data_all.iloc[y])
# 3. Training non-linear SVM
# classifier = SVC(C=1.0, kernel='rbf', random_state=241)
# classifier.fit(data_all, Y)
# def d_int(x, y):
#     return distance.euclidean(data_all.iloc[x], data_all.iloc[y])
```

```
## Application
data_all['arrivalTime'] = range(0, len(data_all))
data_all['exps'] = [[] for _ in range(data_all.shape[0])]
data_all['Rmc'] = [[] for _ in range(data_all.shape[0])]
data_all['isCenter'] = False
data_all['isInCluster'] = False
data_all['center'] = -1
data_all['ev'] = 0
data_all['numberOfSucceeding'] = 0
micro_clusters = {}
PD = []
dataList = []
outlierList = []
outliers = []
timeForProcessingWindow = 0
currentTime = 0
W = 500
slide = 500
k = 55
R = 0.25
data_ev = pd.DataFrame(data=data_all[le])
data_ev['Y_pred'] = 0
slide_cnt = int(math.floor(len(data_all)/slide - W/slide))

# (!) Further 2 executions are presented. Only one should be uncommented
# (e1) result - list of outliers
for i in range(0,slide_cnt+1,1):
    window_start = i*slide
    window_end = i*slide+W
    data_W = data_all[window_start:window_end]
    detectOutlier(data_W, window_end-1, W, slide)
    print('Window ['+str(window_start)+':'+str(window_end)+']')
print('Memory '+str(psutil.Process().memory_info().peak_wset))
print('Time '+str(timeForProcessingWindow))
print('outliers '+str(outliers))
## (e2) result - list of outliers with assigned clusters
# n_cl = 10
# for i in range(0,slide_cnt+1,1):
#     window_start = i*slide
#     window_end = i*slide+W
#     data_W = data_all[window_start:window_end]
```

```
#     print('Window [',window_start,':',window_end,']')
#     detectOutlier(data_W, window_end-1, W, slide)
#     if i >= n_cl:
#         if (i % n_cl == 0):
#             print('-----Trained K-Means-----')
#             ## training K-Means ##
#             pred_1 = data_all[data_ev.Y_pred == 1]
#             pred_1 = pred_1.iloc[:,:le]
#             sil = []
#             if len(pred_1) < 10:
#                 kmax = len(pred_1)-1
#             else:
#                 kmax = 10
#             ran = range(2, kmax)
#             for n_c in ran:
#                 kmeans = KMeans(n_clusters = n_c).fit(pred_1)
#                 labels = kmeans.labels_
#                 sil.append(silhouette_score(pred_1, labels,
#                                             metric = 'euclidean'))
#             cl = ran[np.argmax(sil)]
#             km = KMeans(n_clusters=cl)
#             km.fit(pred_1)
#             km.fit_predict(pred_1)
#             centers = km.cluster_centers_
#             labels = km.labels_
#             ## K-Means statistic ##
#             print('\033[1m'+'Number of datapoints in cluster: '
#                   +'\033[0m'+str(Counter(km.labels_)))
#             print('\033[1m'+'Mean distances:'+'\033[0m')
#             alldistances = km.fit_transform(pred_1)
#             totalDistance = np.min(alldistances, axis=1)
#             for p in np.unique(labels):
#                 subset = totalDistance[(labels == p)]
#                 print(p,' ',subset.mean())
#             clus_stat = pred_1
#             clus_stat['cluster'] = km.labels_
#             clustergrp = clus_stat.groupby('cluster').mean()
#             print ('\033[1m'+'Clustering variable means by cluster:'
#                    +'\033[0m')
#             print(clustergrp)
#             print('')
```

```
#         else:
#             for i in outliers:
#                 if i >= window_start and i <= window_end:
#                     u=km.predict(data_all.iloc[i,:le].
#                                                 values.reshape(1,-1))
#                     print('*Prediction: out ',i,' predicted cluster ',u)
# print('')
# print('----------END----------')
# print('Memory ',psutil.Process().memory_info().peak_wset)
# print('Time ',timeForProcessingWindow)
# print('outliers ',outliers)
```

Approx-Storm

```
import timeit
import sys
import psutil
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from collections import Counter, defaultdict
import matplotlib.lines as mlines

# # Defining algorithm functions
def detectOutlier(data,currentTime,W,slide):
    global timeForProcessingWindow
    global dataList
    global outliers
    startTime = timeit.default_timer()
    # remove expired data from dataList
    if (slide != W):
        if len(dataList) != 0:
            d_lim = -1
            for i in range(0,len(dataList),1):
                d = dataList[i]
                if (d <= currentTime - W):
                    d_lim = i
```

```
            for j in range(d_lim,-1,-1):
                dataList.pop(j)
    else:
        dataList = []

    for dac in data.arrivalTime: #all window
        # do range query for ob
        query = pd.DataFrame(columns = ['col1', 'col2'])
        for nei in data.arrivalTime:
            if dac != nei:
                if (d_int(dac, nei) <= R):
                    query = query.append({'col1' : dac, 'col2' : nei},
                                         ignore_index = True)
        queryResult = []
        for ri in range(0,len(query),1):
            queryResult.append(query.iloc[ri].col2)
        count_before = 0
        for i in range(0,len(queryResult),1):
            # update neighbour for new ob and its neighbour's
            dod = int(queryResult[i])
            if (currentTime <= W):
                if (len(data_all.at[dac,'nn_before']) < k):
                    data_all.iloc[dac].nn_before.append(dod)
            else:
                count_before += 1
            data_all.at[dod,'count_after'] += 1
            # check dod is safe inliers
            if (currentTime > W) and (data_all.at[dod,'count_after'] >= k):
                if (len(safeInlierList) >= W*p):
                    # remove randomly a safe inliers
                    r_index = random.randint(0, len(safeInlierList)-1)
                    remove = safeInlierList[r_index]
                    if r_index in safeInlierList:
                        safeInlierList.remove(r_index)
                    if remove in dataList: dataList.remove(remove)
                    del remove
                if dod not in safeInlierList: safeInlierList.append(dod)
        if (currentTime > W):
            data_all.at[dac,'frac_before']=count_before/len(safeInlierList)
        # store object into dataList
        if dac not in dataList: dataList.append(dac)
```

```
    # Compute number of safe inliers for the first window
    if (currentTime <= W):
        for d in dataList:
            if (data_all.at[d,'count_after'] >= k):
                safeInlierList.append(d)
        # update frac_before for all object in window
        for d in dataList:
            if len(safeInlierList) != 0:
                data_all.at[d,'frac_before'] =
                    len(data_all.iloc[d].nn_before)/len(safeInlierList)

    # (!) further two cases are presented. Only one should be uncommented
    # (c1) add result to outliers (for classical alg and 1st modification)
    for d in dataList:
        # Count preceeding neighbours
        pre = (data_all.at[d,'frac_before'] * (W - currentTime + d))
        if (pre+data_all.at[d,'count_after'] < k) and (d not in outliers):
            outliers.append(d)
            data_ev.at[d,'Y_pred'] = 1
#     # (c2) add result to outliers (for 2-3 modifications)
#     for d in dataList:
#         pre = (data_all.at[d,'frac_before'] * (W - currentTime + d))
#         if (pre+data_all.at[d,'count_after']<k) and (d not in outliers):
#             a = data_all.iloc[x,:9]
#             a = a.values.reshape(1,-1)
#             svm_pred = classifier.predict(a)
#             if (svm_pred[0] != 0):
#                 outliers.append(d)
#                 data_ev.at[d,'Y_pred'] = 1

    timeForProcessingWindow += timeit.default_timer() - startTime
    return outliers

# # For modifications
# (!) Further extra step for 3 modifications.Only one should be uncommented
data_all = data_all[:1000]
le = len(data_all.columns) - 1
Y = data_all[le]
Y = Y.values
data_all = data_all.drop([le], axis = 1)
# 1. Learning weights for weighted distance
```

```
classifier = SVC(C=1.0, kernel='linear', random_state=241)
classifier.fit(data_all, Y)
coef = classifier.coef_
coef = pd.DataFrame(data=coef)
coef = abs(coef)
def d_int(x, y):
    return distance.euclidean(coef.iloc[0,:]*data_all.iloc[x],
                              coef.iloc[0,:]*data_all.iloc[y])
# 2. Training linear SVM
# classifier = SVC(C=1.0, kernel='linear', random_state=241)
# classifier.fit(data_all, Y)
# def d_int(x, y):
#     return distance.euclidean(data_all.iloc[x], data_all.iloc[y])
# 3. Training non-linear SVM
# classifier = SVC(C=1.0, kernel='rbf', random_state=241)
# classifier.fit(data_all, Y)
# def d_int(x, y):
#     return distance.euclidean(data_all.iloc[x], data_all.iloc[y])

# # Application
data_all['arrivalTime'] = range(0, len(data_all))
data_all['exps'] = [[] for _ in range(data_all.shape[0])]
data_all['Rmc'] = [[] for _ in range(data_all.shape[0])]
data_all['isCenter'] = False
data_all['isInCluster'] = False
data_all['center'] = -1
data_all['ev'] = 0
data_all['numberOfSucceeding'] = 0
micro_clusters = {}
PD = []
dataList = []
outlierList = []
outliers = []
timeForProcessingWindow = 0
currentTime = 0
W = 500
slide = 500
p = 0.97
k = 55
R = 0.25
data_ev = pd.DataFrame(data=data_all[le])
```

```
data_ev['Y_pred'] = 0
slide_cnt = int(math.floor(len(data_all)/slide - W/slide))

# (!) Further 2 executions are presented. Only one should be uncommented
# (e1) result - list of outliers
for i in range(0,slide_cnt+1,1):
    window_start = i*slide
    window_end = i*slide+W
    data_W = data_all[window_start:window_end]
    detectOutlier(data_W, window_end-1, W, slide)
    print('Window [',window_start,':',window_end,']')
print('Memory ',psutil.Process().memory_info().peak_wset)
print('Time ',timeForProcessingWindow)
print('outliers ',outliers)
# # (e2) result - list of outliers with assigned clusters
# n_cl = 10
# for i in range(0,slide_cnt+1,1):
#     window_start = i*slide
#     window_end = i*slide+W
#     data_W = data_all[window_start:window_end]
#     print('\033[1m' + 'Window [',window_start,':',window_end,']'
#           + '\033[0m')
#     detectOutlier(data_W, window_end-1, W, slide)
#     if i >= n_cl:
#         if (i % n_cl == 0):
#             print('-----Trained K-Means-----')
#             ## training K-Means ##
#             pred_1 = data_all[data_ev.Y_pred == 1]
#             pred_1 = pred_1.iloc[:,:le]
#             sil = []
#             if len(pred_1) < 10:
#                 kmax = len(pred_1)-1
#             else:
#                 kmax = 10
#             ran = range(2, kmax)
#             for n_c in ran:
#                 kmeans = KMeans(n_clusters = n_c).fit(pred_1)
#                 labels = kmeans.labels_
#                 sil.append(silhouette_score(pred_1, labels,
#                                             metric = 'euclidean'))
#             cl = ran[np.argmax(sil)]
```

```
#                 km = KMeans(n_clusters=cl)
#                 km.fit(pred_1)
#                 km.fit_predict(pred_1)
#                 centers = km.cluster_centers_
#                 labels = km.labels_
#                 ## K-Means statistic ##
#                 print('\033[1m'+'Number of datapoints in cluster: '
#                         +'\033[0m'+str(Counter(km.labels_)))
#                 print('\033[1m'+'Mean distances:'+'\033[0m')
#                 alldistances = km.fit_transform(pred_1)
#                 totalDistance = np.min(alldistances, axis=1)
#                 for p in np.unique(labels):
#                     subset = totalDistance[(labels == p)]
#                     print(p,' ',subset.mean())
#                 clus_stat = pred_1
#                 clus_stat['cluster'] = km.labels_
#                 clustergrp = clus_stat.groupby('cluster').mean()
#                 print ('\033[1m'+'Clustering variable means by cluster:'
#                         +'\033[0m')
#                 print(clustergrp)
#                 print('')
#             else:
#                 for i in outliers:
#                     if i >= window_start and i <= window_end:
#                         u = km.predict(data_all.iloc[i,:9].
#                                                 values.reshape(1,-1))
#                         print('*Prediction: outl ',i,' predicted cluster ',u)
# print('')
# print('----------END----------')
# print('Memory ',psutil.Process().memory_info().peak_wset)
# print('Time ',timeForProcessingWindow)
# print('outliers ',outliers)
```

Abstract-C

```
import timeit
import sys
import psutil
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
```

```
from scipy.spatial import distance
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from collections import Counter, defaultdict
import matplotlib.lines as mlines

# # Defining algorithm functions
def detectOutlier(data,currentTime,W,slide):
    global timeForProcessingWindow
    global dataList
    global outliers
    startTime = timeit.default_timer()
    # remove expired data from dataList
    if (slide != W):
        if len(dataList) != 0:
            d_lim = -1
            for i in range(0,len(dataList),1):
                d = dataList[i]
                if (d <= currentTime - W):
                    d_lim = i
            for j in range(d_lim,-1,-1):
                dataList.pop(j)
    else:
        dataList = []

    # process new slide
    startTime = timeit.default_timer()
    for dac in data.arrivalTime: #all window
        # do range query for ob
        startTime2 = timeit.default_timer()
        interval = currentTime+1 - W
        i = 1
        # creating empty lt_cnt for all upcoming windows
        while dac >= interval:
            if (len(data_all.at[dac,'lt_cnt']) == 0) or (
                len(data_all.at[dac,'lt_cnt']) < i):
                data_all.iloc[dac].lt_cnt.append(0)
            i += 1
            interval = interval + slide
        query = pd.DataFrame(columns = ['col1', 'col2'])
```

```
        for nei in data.arrivalTime:
            if dac != nei:
                if (d_int(dac, nei) <= R):
                    query = query.append({'col1' : dac, 'col2' : nei},
                                                    ignore_index = True)
        if (currentTime+1 == W):
            for ri in range(0,len(query),1):
                ind1 = int(query.at[ri,'col1'])
                ind2 = int(query.at[ri,'col2'])
                for n in range(0,len(data_all.at[ind1,'lt_cnt']),1):
                    if (ind2 >= currentTime+1-W+n*slide):
                        data_all.iloc[ind1].lt_cnt[n] += + 1
        elif (currentTime > W):
            for ri in range(0,len(query),1):
                ind1 = int(query.at[ri,'col1'])
                ind2 = int(query.at[ri,'col2'])
                for n in range(0,len(data_all.at[ind1,'lt_cnt']),1):
                    if (ind2 >= currentTime-W) and (
                        ind2 >= currentTime+1-W+n*slide):
                        data_all.iloc[ind1].lt_cnt[n] += + 1
        startTime3 = timeit.default_timer()
        if dac not in dataList: dataList.append(dac)
        psutil.virtual_memory().percent

    # (!) further two cases are presented. Only one should be uncommented
    # (c1) add result to outliers (for classical alg and 1st modification)
    for d in dataList:
        p = data_all.iloc[d]
        if len(p.lt_cnt) != 0:
            if (p.lt_cnt[0] < k) and (d not in outliers):
                outliers.append(d)
                   data_ev.at[d,'Y_pred'] = 1
            data_all.iloc[d].lt_cnt.pop(0)
#     # (c2) add result to outliers (for 2-3 modifications)
#     for d in dataList:
#         p = data_all.iloc[d]
#         if len(p.lt_cnt) != 0:
#             if (p.lt_cnt[0] < k) and (d not in outliers):
#                 a = data_all.iloc[x,:le]
#                 a = a.values.reshape(1,-1)
#                 svm_pred = classifier.predict(a)
```

```
#                     if (svm_pred[0] != 0):
#                         outliers.append(d)
#                         data_ev.at[d,'Y_pred'] = 1
#                 data_all.iloc[d].lt_cnt.pop(0)

    timeForProcessingWindow += timeit.default_timer() - startTime
    return outliers

## For modifications
# (!) Further extra step for 3 modifications.Only one should be uncommented
data_all = data_all[:1000]
le = len(data_all.columns) - 1
Y = data_all[le]
Y = Y.values
data_all = data_all.drop([le], axis = 1)
# 1. Learning weights for weighted distance
classifier = SVC(C=1.0, kernel='linear', random_state=241)
classifier.fit(data_all, Y)
coef = classifier.coef_
coef = pd.DataFrame(data=coef)
coef = abs(coef)
def d_int(x, y):
    return distance.euclidean(coef.iloc[0,:]*data_all.iloc[x],
                                        coef.iloc[0,:]*data_all.iloc[y])
# 2. Training linear SVM
# classifier = SVC(C=1.0, kernel='linear', random_state=241)
# classifier.fit(data_all, Y)
# def d_int(x, y):
#     return distance.euclidean(data_all.iloc[x], data_all.iloc[y])
# 3. Training non-linear SVM
# classifier = SVC(C=1.0, kernel='rbf', random_state=241)
# classifier.fit(data_all, Y)
# def d_int(x, y):
#     return distance.euclidean(data_all.iloc[x], data_all.iloc[y])

## Application
data_all['arrivalTime'] = range(0, len(data_all))
data_all['exps'] = [[] for _ in range(data_all.shape[0])]
data_all['Rmc'] = [[] for _ in range(data_all.shape[0])]
data_all['isCenter'] = False
data_all['isInCluster'] = False
```

```
data_all['center'] = -1
data_all['ev'] = 0
data_all['numberOfSucceeding'] = 0
micro_clusters = {}
PD = []
dataList = []
outlierList = []
outliers = []
timeForProcessingWindow = 0
currentTime = 0
W = 500
slide = 500
k = 55
R = 0.25
data_ev = pd.DataFrame(data=data_all[le])
data_ev['Y_pred'] = 0
slide_cnt = int(math.floor(len(data_all)/slide - W/slide))

# (!) Further 2 executions are presented. Only one should be uncommented
# (e1) result - list of outliers
for i in range(0,slide_cnt+1,1):
    window_start = i*slide
    window_end = i*slide+W
    data_W = data_all[window_start:window_end]
    detectOutlier(data_W, window_end-1, W, slide)
    print('Window [',window_start,':',window_end,']')
print('Memory ',psutil.Process().memory_info().peak_wset)
print('Time ',timeForProcessingWindow)
print('outliers ',outliers)
# # (e2) result - list of outliers with assigned clusters
# n_cl = 10
# for i in range(0,slide_cnt+1,1):
#     window_start = i*slide
#     window_end = i*slide+W
#     data_W = data_all[window_start:window_end]
#     print('\033[1m' + 'Window [',window_start,':',window_end,']'
#           + '\033[0m')
#     detectOutlier(data_W, window_end-1, W, slide)
#     if i >= n_cl:
#         if (i % n_cl == 0):
#             print('-----Trained K-Means-----')
```

```
#             ## training K-Means ##
#             pred_1 = data_all[data_ev.Y_pred == 1]
#             pred_1 = pred_1.iloc[:,:le]
#             sil = []
#             if len(pred_1) < 10:
#                 kmax = len(pred_1)-1
#             else:
#                 kmax = 10
#             ran = range(2, kmax)
#             for n_c in ran:
#                 kmeans = KMeans(n_clusters = n_c).fit(pred_1)
#                 labels = kmeans.labels_
#                 sil.append(silhouette_score(pred_1, labels,
#                                 metric = 'euclidean'))
#             cl = ran[np.argmax(sil)]
#             km = KMeans(n_clusters=cl)
#             km.fit(pred_1)
#             km.fit_predict(pred_1)
#             centers = km.cluster_centers_
#             labels = km.labels_
#             ## K-Means statistic ##
#             print('\033[1m'+'Number of datapoints in cluster: '
#                     +'\033[0m'+str(Counter(km.labels_)))
#             print('\033[1m'+'Mean distances:'+'\033[0m')
#             alldistances = km.fit_transform(pred_1)
#             totalDistance = np.min(alldistances, axis=1)
#             for p in np.unique(labels):
#                 subset = totalDistance[(labels == p)]
#                 print(p,'  ',subset.mean())
#             clus_stat = pred_1
#             clus_stat['cluster'] = km.labels_
#             clustergrp = clus_stat.groupby('cluster').mean()
#             print ('\033[1m'+'Clustering variable means by cluster:'
#                     +'\033[0m')
#             print(clustergrp)
#             print('')
#         else:
#             for i in outliers:
#                 if i >= window_start and i <= window_end:
#                     u = km.predict(data_all.iloc[i,:le].
#                                         values.reshape(1,-1))
```

```
#                       print('*Prediction: outl ',i,' predicted cluster ',u)
# print('')
# print('----------END----------')
# print('Memory ',psutil.Process().memory_info().peak_wset)
# print('Time ',timeForProcessingWindow)
# print('outliers ',outliers)
```

MCOD

```
import timeit
import sys
import psutil
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from collections import Counter, defaultdict
import matplotlib.lines as mlines

# # Defining algorithm functions
# Additional functions
def resetObject(o):
    arr = o.arrivalTime
    data_all.at[arr,'exps'] = []
    data_all.at[arr,'Rmc'] = []
    data_all.at[arr,'isCenter'] = False
    data_all.at[arr,'isInCluster'] = False
    data_all.at[arr,'ev'] = 0
    data_all.at[arr,'center'] = -1
    data_all.at[arr,'numberOfSucceeding'] = 0

def isSameSlide(o1, o2):
    if math.floor(o1.arrivalTime/slide)==math.floor(o2.arrivalTime/slide):
        return 0 #same slide
    elif math.floor(o1.arrivalTime/slide)<math.floor(o2.arrivalTime/slide):
        return -1 #o1 before o2
    else:
        return 1 #o1 after o2
```

```
def checkInlier(p):
    arr = p.arrivalTime
    data_all.at[arr,'exps'] = sorted(data_all.at[arr,'exps'])
    while (len(data_all.at[arr,'exps']) > k
            - data_all.at[arr,'numberOfSucceeding']) and (
            len(data_all.at[arr,'exps']) > 0):
        del data_all.iloc[arr].exps[0]
    if (len(data_all.at[arr,'exps']) > 0):
        data_all.at[arr,'ev'] = data_all.iloc[arr].exps[0]
    else:
        data_all.at[arr,'ev'] = 0
    if (len(data_all.at[arr,'exps'])
            + data_all.at[arr,'numberOfSucceeding'] >= k):
        if arr in outlierList: outlierList.remove(arr)
    else:
        if arr not in outlierList: outlierList.append(arr)

def isOutlier(d):
    arr = d.arrivalTime
    result = False
    if data_all.at[arr,'numberOfSucceeding']
      + len(data_all.at[arr,'exps']) < k:
        result = True
    return result

def isNeighbour(p, o):
    result = False
    distance = d_int(p.arrivalTime, o.arrivalTime)
    if distance <= R:
        result = True
    return result

# Functions for search
def findClusterIn3_2Range(d):
    result = []
    if micro_clusters:
        for center_id in micro_clusters.keys():
            # compute the distance
            distance = d_int(center_id, d.arrivalTime)
            if (distance <= R * 3.0 / 2):
```

```
                result.append(center_id)
    return result

def findNearestCenter(d):
    min_distance = sys.float_info.max
    min_center_id = -1
    for center_id in micro_clusters.keys():
        # compute the distance
        distance = d_int(center_id, d.arrivalTime)
        if (distance < min_distance):
            min_distance = distance
            min_center_id = center_id
    return min_center_id

def findNeighbourR2InPD(d):
    result = []
    for p in PD:
        if p != d.arrivalTime:
            distance = d_int(p, d.arrivalTime)
            if (distance <= R*1.0/2):
                result.append(p)
    return result

def findNeighbourInR3_2InPD(d):
    result = []
    for p in PD:
        if p != d.arrivalTime:
            distance = d_int(p, d.arrivalTime)
            if (distance <= R*3.0/2):
                result.append(p)
    return result

# Adding objects
def formNewCluster(d, neighboursInR2Distance):
    arr_d = d.arrivalTime
    data_all.at[arr_d,'isCenter'] = True
    data_all.at[arr_d,'isInCluster'] = True
    data_all.at[arr_d,'center'] = arr_d
    for p in neighboursInR2Distance:
        if p in PD: PD.remove(p)
        if (isOutlier(data_all.iloc[p])):
```

```
                outlierList.remove(p)
            resetObject(data_all.iloc[p])
            data_all.at[p,'isInCluster'] = True
            data_all.at[p,'center'] = arr_d
            data_all.at[p,'isCenter'] = False
        # add center to neighbour list
        sorted(neighboursInR2Distance)
        micro_clusters[arr_d] = neighboursInR2Distance
        list_rmc = findNeighbourInR3_2InPD(d)
        for o in list_rmc:
            if (isNeighbour(data_all.iloc[o], d)):
                if (isSameSlide(data_all.iloc[o], d) <= 0):
                    data_all.at[o,'numberOfSucceeding'] += 1
                else:
                    data_all.iloc[o].exps.append(arr_d + W)
                checkInlier(data_all.iloc[o])
            data_all.iloc[o].Rmc.append(arr_d)

    def addToCluster(nearest_center_id, d):
        arr_d = d.arrivalTime
        # update for points in cluster
        data_all.at[arr_d,'isCenter'] = False
        data_all.at[arr_d,'isInCluster'] = True
        data_all.at[arr_d,'center'] = nearest_center_id
        cluster = micro_clusters[nearest_center_id]
        cluster.append(arr_d)
        micro_clusters.update({ nearest_center_id : cluster})
        # update for points in PD which Rmc list contains center
        for p in PD:
            # check if inPD is neighbour of d
            if nearest_center_id in data_all.at[p,'Rmc']:
                distance = d_int(arr_d, p)
                if (distance <= R):
                    if (isSameSlide(d, data_all.iloc[p]) <= 0):
                        data_all.at[arr_d,'numberOfSucceeding'] += 1
                    else:
                        data_all.iloc[arr_d].exps.append(d.arrivalTime + W)
                    checkInlier(data_all.iloc[p])

    def addToPD(o, fromCluster):
        arr_o = o.arrivalTime
```

```
    for p in PD:
        # compute distance
        distance = d_int(arr_o, p)
        if (distance <= R):
            # check inPD is succeeding or preceding neighbour
            if (isSameSlide(data_all.iloc[p], o) == -1):
                data_all.iloc[arr_o].exps.append(data_all.
                            at[p,'arrivalTime'] + W)
                if (not fromCluster):
                    data_all.at[p,'numberOfSucceeding'] += 1
            elif (isSameSlide(data_all.iloc[p], o) == 0):
                data_all.at[arr_o,'numberOfSucceeding'] += 1
                if (not fromCluster):
                    data_all.at[p,'numberOfSucceeding'] += 1
            else:
                data_all.at[arr_o,'numberOfSucceeding'] += 1
                if (not fromCluster):
                    data_all.iloc[p].exps.append(arr_o + W)
            if (not fromCluster):
                checkInlier(data_all.iloc[p])
    # find neighbours in clusters (3R/2)
    clusters = findClusterIn3_2Range(o)
    for i in clusters:
        clust = micro_clusters[i]
        for p in clust:
            if isNeighbour(data_all.iloc[p], o):
                if (isSameSlide(o, data_all.iloc[p]) <= 0):
                    # o is preceeding neighbour
                    data_all.at[arr_o,'numberOfSucceeding'] += 1
                else:
                    # p is preceeding neighbour
                    data_all.iloc[arr_o].exps.append(
                        data_all.iloc[p].arrivalTime + W)
    checkInlier(o)
    if arr_o not in PD:
        PD.append(arr_o)

# Removing objects
def removeFromCluster(d):
    # get the cluster
    cluster = micro_clusters[d.center]
```

```
    if len(cluster) != 0:
        if (d.center != d.arrivalTime):
            cluster.remove(d.arrivalTime)
        micro_clusters[d.center] = cluster
        if (len(cluster) < k) or (d.center == d.arrivalTime):
            # remove this cluster from micro cluster list
            del micro_clusters[d.center]
            cluster = sorted(cluster)
            # process the center of cluster
            resetObject(data_all.iloc[d.center])
            data_all.at[d.center,'numberOfSucceeding'] += len(cluster)-1
            addToPD(data_all.iloc[d.center], True)
            # process the objects in clusters
            for i in range(0,len(cluster),1):
                o = cluster[i]
                # reset all objects
                resetObject(data_all.iloc[o])
                # put into PD
                data_all.at[o,'numberOfSucceeding'] += len(cluster)-1-i
                addToPD(data_all.iloc[o], True)

def removeFromPD(d):
    # remove from PD
    if d.arrivalTime in PD:
        PD.remove(d.arrivalTime)
    if (d.numberOfSucceeding + len(d.exps) < k):
        if d.arrivalTime in outlierList:
            outlierList.remove(d.arrivalTime)
    for p in outlierList:
        while (len(data_all.at[p,'exps']) > 0) and (
                data_all.iloc[p].exps[0] <= d.arrivalTime + W):
            data_all.at[p,'exps'].pop(0)
            if len(data_all.at[p,'exps']) == 0:
                data_all.at[p,'ev'] = 0
            else:
                data_all.at[p,'ev'] = data_all.iloc[p].exps[0]

# Outlier detection
def processNewData(d):
    # add data point to datalist
    global dataList
```

```
        if d.arrivalTime not in dataList:
            dataList.append(d.arrivalTime)
        nearest_center_id = findNearestCenter(d)
        if d.arrivalTime != nearest_center_id:
            min_distance = sys.float_info.max
            if (nearest_center_id > -1):
                # found neareast cluster
                min_distance = d_int(nearest_center_id, d.arrivalTime)
            # assign to cluster if min distance <= R/2
            if (min_distance <= R/2) and (
                d.arrivalTime not in micro_clusters[nearest_center_id]):
                addToCluster(nearest_center_id, d)
            else:
                # find all neighbours for d in PD that can  form a cluster
                neighboursInR2Distance = findNeighbourR2InPD(d)
                if (len(neighboursInR2Distance) >= k):
                    # form new cluster
                    formNewCluster(d, neighboursInR2Distance)
                else:
                    # cannot form a new cluster
                    addToPD(d, False)

def detectOutlier(data,currentTime,W,slide):
    global outliers
    global dataList
    global outlierList
    global PD
    global micro_clusters
    global timeForProcessingWindow
    startTime = timeit.default_timer()
    if slide != W:
        # purge expired object
        if len(dataList) != 0:
            d = dataList[0]
            while data_all.at[d,'arrivalTime'] <= currentTime - W:
                # remove d from data List
                dataList.pop(0)
                # if point in cluster
                if (data_all.at[d,'isInCluster']):
                    removeFromCluster(data_all.iloc[d])
                # if point in PD
```

```
                if d in PD:
                    removeFromPD(data_all.iloc[d])
                resetObject(data_all.iloc[d])
                d = dataList[0]
    else:
        micro_clusters = {}
        dataList = []
        PD = []
        outlierList = []

    # process new data
    for i in range(0,len(data),1):
        processNewData(data.iloc[i])

    # (!) further two cases are presented. Only one should be uncommented
    # (c1) add result to outliers (for classical alg and 1st modification)
    for x in outlierList:
        if (x not in outliers):
            outliers.append(x)
            data_ev.at[x,'Y_pred'] = 1
#     # (c2) add result to outliers (for 2-3 modifications)
#     for x in outlierList:
#         if (x not in outliers):
#             a = data_all.iloc[x,:9]
#             a = a.values.reshape(1,-1)
#             svm_pred = classifier.predict(a)
#             if (svm_pred[0] != 0):
#                 outliers.append(x)
#                 data_ev.at[x,'Y_pred'] = 1

    timeForProcessingWindow += timeit.default_timer() - startTime
    return outliers

# # For modifications
# (!) Further extra step for 3 modifications.Only one should be uncommented
data_all = data_all[:1000]
le = len(data_all.columns) - 1
Y = data_all[le]
Y = Y.values
data_all = data_all.drop([le], axis = 1)
# 1. Learning weights for weighted distance
```

```
classifier = SVC(C=1.0, kernel='linear', random_state=241)
classifier.fit(data_all, Y)
coef = classifier.coef_
coef = pd.DataFrame(data=coef)
coef = abs(coef)
def d_int(x, y):
    return distance.euclidean(coef.iloc[0,:]*data_all.iloc[x],
                                        coef.iloc[0,:]*data_all.iloc[y])
# 2. Training linear SVM
# classifier = SVC(C=1.0, kernel='linear', random_state=241)
# classifier.fit(data_all, Y)
# def d_int(x, y):
#     return distance.euclidean(data_all.iloc[x], data_all.iloc[y])
# 3. Training non-linear SVM
# classifier = SVC(C=1.0, kernel='rbf', random_state=241)
# classifier.fit(data_all, Y)
# def d_int(x, y):
#     return distance.euclidean(data_all.iloc[x], data_all.iloc[y])

# # Application
data_all['arrivalTime'] = range(0, len(data_all))
data_all['exps'] = [[] for _ in range(data_all.shape[0])]
data_all['Rmc'] = [[] for _ in range(data_all.shape[0])]
data_all['isCenter'] = False
data_all['isInCluster'] = False
data_all['center'] = -1
data_all['ev'] = 0
data_all['numberOfSucceeding'] = 0
micro_clusters = {}
PD = []
dataList = []
outlierList = []
outliers = []
timeForProcessingWindow = 0
currentTime = 0
W = 500
slide = 500
k = 55
R = 0.25

data_ev = pd.DataFrame(data=data_all[le])
```

```
data_ev['Y_pred'] = 0
slide_cnt = int(math.floor(len(data_all)/slide - W/slide))

# (!) Further 2 executions are presented. Only one should be uncommented
# (e1) result - list of outliers
for i in range(0,slide_cnt+1,1):
    window_start = i*slide
    window_end = i*slide+W
    data_W = data_all[window_start:window_end]
    detectOutlier(data_W, window_end-1, W, slide)
    print('Window [',window_start,':',window_end,']')
# # (e2) result - list of outliers with assigned clusters
# n_cl = 10
# for i in range(0,slide_cnt+1,1):
#     window_start = i*slide
#     window_end = i*slide+W
#     data_W = data_all[window_start:window_end]
#     print('\033[1m' + 'Window [',window_start,':',window_end,']'
#           + '\033[0m')
#     detectOutlier(data_W, window_end-1, W, slide)
#     if i >= n_cl:
#         if (i % n_cl == 0):
#             print('-----Trained K-Means-----')
#             ## training K-Means ##
#             pred_1 = data_all[data_ev.Y_pred == 1]
#             pred_1 = pred_1.iloc[:,:9]
#             sil = []
#             if len(pred_1) < 10:
#                 kmax = len(pred_1)-1
#             else:
#                 kmax = 10
#             ran = range(2, kmax)
#             for n_c in ran:
#                 kmeans = KMeans(n_clusters = n_c).fit(pred_1)
#                 labels = kmeans.labels_
#                 sil.append(silhouette_score(pred_1, labels,
#                                             metric = 'euclidean'))
#             cl = ran[np.argmax(sil)]
#             km = KMeans(n_clusters=cl)
#             km.fit(pred_1)
#             km.fit_predict(pred_1)
```

```
#                centers = km.cluster_centers_
#                labels = km.labels_
#                ## K-Means statistic ##
#                print('\033[1m'+'Number of datapoints in cluster: '
#                        +'\033[0m'+str(Counter(km.labels_)))
#                print('\033[1m'+'Mean distances:'+'\033[0m')
#                alldistances = km.fit_transform(pred_1)
#                totalDistance = np.min(alldistances, axis=1)
#                for p in np.unique(labels):
#                    subset = totalDistance[(labels == p)]
#                    print(p,' ',subset.mean())
#                clus_stat = pred_1
#                clus_stat['cluster'] = km.labels_
#                clustergrp = clus_stat.groupby('cluster').mean()
#                print ('\033[1m'+'Clustering variable means by cluster:'
#                         +'\033[0m')
#                print(clustergrp)
#                print('')
#            else:
#                for i in outliers:
#                    if i >= window_start and i <= window_end:
#                        u = km.predict(data_all.iloc[i,:le].
#                                                values.reshape(1,-1))
#                        print('*Prediction: outl ',i,' predicted cluster ',u)

print('')
print('----------END----------')
print('Memory ', psutil.Process().memory_info().peak_wset)
print('Time ', timeForProcessingWindow)
print('outliers ', outliers)
```