ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Real-time Anomaly Detection at Scale

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΝΙΚΟΛΑΟΥ ΓΑΒΑΛΑ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
Αθήνα, Μάρτιος 2019

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Real-time Anomaly Detection at Scale

Διπλωματική Εργασία

του

ΝΙΚΟΛΑΟΥ ΓΑΒΑΛΑ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Μαρτίου 2019.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
....	....	....
Νεκτάριος Κοζύρης	Δημήτριος Τσουμάκος	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Αν.Καθηγητής Ι.Π.	Επ.Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2019

(Υπογραφή)

....

Γαβαλας Νικολαος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών
Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Abstract

Anomaly Detection is a field of Machine Learning used by systems to identify observations that differ from the majority of data and are often linked directly to unexpected behaviour, errors, and other forms of novelties.

Common applications of Anomaly Detection include but are not limited to credit card fraud detection, machinery or computer behaviour monitoring, network intrusion detection, real-time analytics, etc.

In this thesis we study algorithms and methods for Anomaly Detection that enable identification of outliers both in real-time, in order to prevent unwanted events as soon as possible, and at a big scale, since the volume of data in our era is growing exponentially.

Keywords

Machine Learning, Anomaly Detection, Real-Time Systems, Distributed Stream Processing, Big Data

Acknowledgements

I would like to thank my professor Nectarios Koziris, for giving me the opportunity to explore and thoroughly study Distributed Systems and Machine Learning in this work. I would also like to thank the postdoctoral researcher Katerina Doka for her guidance during the writing of this thesis. Last but not least, I express my gratitude to my family and friends who supported my throughout this journey.

Contents

List of tables

List of figures

Chapter 1

Introduction

1.1 Motivation

Anomaly Detection (or Outlier Detection) is a field of Machine Learning with many important applications. It is used by systems to identify items or observations with peculiar characteristics, which cause them to differ from the majority of data. The detection of such anomalies translates directly to detection of unexpected behaviour, errors, and other forms of novelties.

The applications are endless; credit card fraud detection, machinery or computer behaviour monitoring, network intrusion detection, analytics, etc.

However, when applying Anomaly Detection, it is crucial to leverage computation methods and engineering principles that enable detection at a big scale, as nowadays the amount of data produced daily is massive and the growth rate is exponential.

Also, the nature of Anomalies (which are directly related to unwanted events), compels us to minimize the latency of Anomaly Detection Systems, so that action can be taken as soon as possible. This requirement has led to the development of algorithms and systems that operate in real-time, and process the data in an "online", or "streaming" manner, meaning that processing is done on-the-fly, as soon as data is collected, in contrast to the more classic "batch" way of processing, which requires a-priori knowledge of the whole dataset.

1.2 Objectives

The objective of the current thesis is to study anomaly detection methods and algorithms, then implement the most suitable of them to operate in real-time and in an on-line manner, and finally perform tests and examine the characteristics and capabilities of each of them, in the context of Big Data.

For the implementation we have to make use of a Distributed Stream Processing Framework (Apache Flink), which will enable us to perform our computations on a computational cluster of machines instead of a single processor, and also process the data as a stream. This way we are able to achieve high throughput (thanks to the high parallelization of distributed computing) while also keeping the system latency to a minimum.

1.3 Thesis Outline

In Chapter 2 we will define the fundamental terminology and conventions used throughout this work. We also briefly present some of the most well known algorithms for Anomaly Detection, what stream and batch processing is, and what stream processing framework we will use. In Chapter 3 we present the four algorithms that met our requirements, we explain why we chose them and what are their characteristics, and we also provide some implementation details. In Chapter 4, we carry out experiments using the algorithms, and examine how they behave on different datasets, both synthetic and real, and what is their performance when executed on a single machine and on a cluster. We also present a convenient way we used to automate the deployment of our cluster. In the final Chapter, 5, we summarize the current work's findings and recommend some possible extensions that would make the project an integrated production-ready Anomaly Detection system for Big Data.

Chapter 2

Background

2.1 Machine Learning and Anomaly Detection

Machine learning, a subset of Artificial Intelligence, is the study of algorithms and statistical models that computer systems use, to effectively perform a specific task without being explicitly programmed [5]. Machine learning algorithms build a mathematical model based on sample data, and make predictions or decisions relying on patterns and inference.

The process of the mathematical model creation is called *training*, and the sample data used for this cause are called *training data*.

The most commonly used "structure" to represent data, and the one we will use throughout this thesis, is that data are comprised by a number of *examples* or *instances*, the total number of which is the *length* or *size* of the dataset, which are basically vectors (one-dimensional arrays). These vectors are of the same length m, also called the dimensionality of the dataset. The values of the elements of each of these vectors correspond to the values of the respective *example* , for each of a total of m *features*. Therefore, we can say that our data forms a matrix with dimensions $n \times m$, n being the size of the dataset.

There are three Machine Learning paradigms, regarding the way of "learning" or training:

- Supervised Learning, in which the ML model adjusts its parameters by getting feedback using a labelled dataset, .i.e. a training dataset whose examples are accompanied by a "correct" or "expected" value,
- Unsupervised Learning, for when the dataset is unlabelled,
- Reinforcement Learning, that uses a different learning approach involving software *agents* and *actions* in an *environment* in order to maximize a cumulative

reward.

The most common applications of Machine Learning are *Classification* and *Regression*, which use Supervised Learning methods, *Clustering*, which uses Unsupervised Learning methods, and *Anomaly Detection*, for which there are Supervised, Semi-Supervised and Unsupervised methods.

One of the goals of the current work is to explore one of these particular categories, Anomaly Detection.

Anomaly Detection, as the name suggests, is the identification of observations that deviate from normal ones. These deviating observations are called *anomalies* or *outliers*. According to Aggarwal [4], an outlier is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism. We care about identifying anomalies as soon as possible, because they are almost always linked to unwanted events.

2.2 Overview of Anomaly Detection Algorithms

According to [8], the Anomaly Detection techniques are:

- Classification based
  - Neural Networks-based
  - Bayesian Networks-based
  - Support Vector Machine (SVM)-based
  - Rule based
- Nearest-Neighbor-based
- Clustering-based
- Statistical
- Information theoretic
- Spectral

, to which we will add the Ensemble-based algorithms.

In the next few subsections, we will explore the characteristics of some algorithms, and choose those that seem the most promising according to our goals and we will later implement and evaluate in Chapters 3 and 4.

Neural Networks-based methods

Neural Networks are widely used nowadays mainly for Classification purposes, and have also been studied in the context of Anomaly Detection. They are very effective for problems with complex non-linear hypotheses and many features.

A Neural Network consists of layers of interconnected artificial neurons, like the ones shown in figure 2.1. Each neuron receives the inputs to the system (or inputs from neighboring neurons, in the case of neurons found in deeper layers), applies some weights to each input, aggregates them, and maps them to an output through an *activation function*.

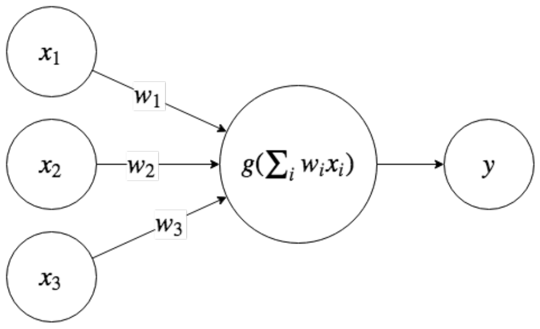


Figure 2.1: Artificial Neuron, the building block of a Neural Network

A Neural Network gives predictions by propagating its inputs through all its layers of neurons and hence produces one (or more) outputs. This process is called *feed-forward*. The training of a Neural Network is basically the adjustment of its weights according to a (labelled) training set. One common method for this task is by defining a *cost function*, for estimating how far are the outputs or predictions of the Neural Network compared to the actual ones, and then by readjusting the weights "following" the gradient of this cost function, with respect to the weights. This gradient is found by computing the *errors*, .i.e. the distances of the predicted outputs from the actual ones, and propagating them back through the network, towards the input. This process is called *back-propagation* and the optimization method is called *gradient descent*.

To apply for Anomaly Detection, the most simple way would be to use an output layer of one neuron, the output of which would be a binary descriptor, 1 for anomalies, 0 for normal observations, but there more effective methods that this one.

An interesting method was proposed in [12]. The authors used an "Elman Recurrent Network", which showed better results compared to the standard multi-layered perceptron we described above. The Elman RNN, is a three-layered Neural Network, which, additionally to the neurons, has a set of *context units*. Each context unit's inputs are the outputs of the neurons of a layer, and the nodes' outputs are fed back to the same neurons of the same layer. This recurrent layout allows the RNN to retain information between inputs. The measure of anomaly of each sequential event is the difference between the output at time i and the input at time $i + 1$. This approach yields good experimental results, but it suits better for time-series Anomaly Detection, i.e. for Anomaly Detection in series of data whose sequence and ordering matters, and not for all cases.

Another method proposes the use of a Replicator Neural Network [13], a variant of the classic multi-layered perceptron, where the input and output layers have the same size, but the hidden layers have smaller sizes. During training, the Replicator Neural Network is fed with normal data, and is required to reproduce the input at the output layer, which forces the hidden layers to learn a compressed version of the input. Then, during inference, the MSE (mean square error) between the produced output and the input is calculated. If the MSE is low, then the example was probably normal. But if the MSE is high, the data point is most likely an anomaly.

The downside of using Neural Networks though is that they require a lot of data to be trained at a satisfactory degree, and most importantly, they have notoriously heavy computational costs, which have even led to introduction of new special hardware.

Bayesian Networks

A Bayesian Network is a probabilistic graphical model, that represents a set of variables and their conditional dependencies via a DAG (directed acyclic graph) and conditional probability tables (CPT). Due to their ability to represent causal relationships, they can be used to predict consequences of actions.

Bayesian Networks have been employed for Anomaly Detection, mostly in the form of dynamic Bayesian Networks, whose difference is that they "evolve" over time. In [14], the authors deployed a dynamic Bayesian Network for this cause but their approach is constructed for two streams, and they state that adding more streams is a non-trivial task. Hence, the approach is not really suitable for handling a large amount of streams.

Ensemble-based algorithms

This category includes algorithms that employ an *ensemble* of weak classifiers based on random number generation, which are then averaged. If the ensemble is big enough, the negative effects of randomness are cancelled out and the classification capabilities are strong.

Such algorithms are the Isolation Forest, Half-Space Trees, and the Lightweight

On-line Detector of Anomalies, all of which were selected and implemented, with details in Chapter 3. The reasons why they were selected are presented in the same chapter.

Nearest-Neighbor-based methods

Nearest-Neighbor methods are data-centric methods, meaning that they don't use a model. The most popular in this category is the KNN algorithm, k-nearest neighbor, which has been used extensively for Classification, and has later been used in outlier identification [18]. The anomaly scores assigned to a sample is based on the distance to its k-th nearest neighbor.

KNN It has received criticism for not being able to detect outliers in data with clusters of different densities. Later, LOF [6] was introduced (Local Outlier Factor), which solved this problem, and yielded better results. Another notable method falling in this category is the Stochastic Outlier Detection algorithm [15].

However, despite being quite effective sometimes, KNN-based methods have a major drawback; the anomaly score they assign to samples is driven by the nearest neighbor search, which is an operation with $O(n)$ time, making the algorithms have a total time complexity of $O(n^2)$. Such complexities are absolutely forbidding for large scale applications.

Support Vector Machine (SVM) methods

Support Vector Machines is a powerful Machine Learning method used mostly for Classification, but it has also been applied to Anomaly Detection, especially the One-Class SVM (or 1-SVM).

The original SVM is a solution to the two-class classification problem. It works by using two parallel hyperplanes that separate the data in two classes, so that the distance between them is as large as possible. The region bounded by these two hyperplanes is called *margin*, and the maximum-margin hyperplane is the hyperplane that lies halfway between them. Finding this maximum-margin hyperplane, which is then used to classify data points, is an optimization problem solved using Lagrange Multipliers.

The linearity of the hyperplane can be relaxed and non-linear kernels can be used too. Nevertheless, the kernel needs to be picked beforehand and it can be difficult to decide which kernel suits the data best. Also, SVM methods are super-linear, with complexities of $O(n^3d)$, n being the size of the dataset and d the number of features.

Statistical techniques

In statistical based approaches, the aim is to learn a statistical model for a normal behaviour of a dataset. Thereafter, the observations that are not (or have low probability to) fit into that model are marked as outliers. Usually the statistical model is a particular distribution, the parameters of which are estimated using Maximum Likelihood Estimates (MLE).

The downside of these algorithms is that a priori knowledge regarding the underlying distribution of the dataset is required, which is not always available [24].

In terms of data streams, the authors of [28] have used a Gaussian Mixture Model to assign anomaly scores to the incoming data points.

An algorithm from this category (Multivariate Gaussian) was selected and all the relevant details are presented in Chapter 3.

Spectral techniques

Spectral techniques try to find an approximation of the data using a combination of attributes that capture the bulk of the variability in the data [8]. Such techniques are based on the assumption that data can be embedded into a lower dimensional subspace in which normal instances and anomalies appear significantly different.

Thus the general approach adopted by spectral anomaly detection techniques is to determine such subspaces (embeddings, projections, etc.) in which the anomalous instances can be easily identified. Such techniques can work in an unsupervised as well as a semisupervised setting.

Several techniques use Principal Component Analysis (PCA) for projecting data into a lower dimensional space. A normal instance that satisfies the correlation structure of the data will have a low value for such projections while an anomalous instance that deviates from the correlation structure will have a large value.

These techniques however are only useful when the features of the dataset are actually correlated, and projection to a lower dimensional space is therefore feasible. Also, the complexity of their training is often $O(nd^3)$ or $O(nd^4)$, where d is the number of features, making the techniques unusable to highly dimensional data.

When not used for Anomaly Detection by themselves, PCA and other related algorithms are very useful for supporting other algorithms by reducing the dimensionality of the data, which not only speeds up the program execution (for all algorithms), but can also make some algorithms detect anomalies with greater precision.

2.3 Batch vs Stream Processing

Now there are two ways of data processing, *batch* processing and *stream* processing, each one having advantages and disadvantages and being more suitable to some applications, when compared to the other.

Batch processing is the more "traditional" way, in which the dataset usually "sits" in a database (or storage system in general), and is processed "as a whole". This allows for more flexibility to build complex algorithms, which are usually the most effective. Nevertheless, it is generally slow, and unavoidably adds a big "lag" between the time the data is collected, until the time that they have been processed and meaningful results have been extracted from them.

On the other hand we have stream processing. A stream of data is a continuous flow of data instances, that theoretically never ends (has infinite length), and passes only one time through a stream processing system, before a prediction or decision is made based on this particular example and the model. The model may have been built on batch data first and loaded for inference, or built entirely on-line. On-line models are updated by every instance passing through.

With this way of processing, the time required to extract information from the data, since their acquisition, also know as *latency*, is minimized to the point where we can talk about "real-time" processing. Stream processing is absolutely mandatory for some types of applications because of their very nature, such as real-time analytics, event processing, machine monitoring, network intrusion detection, credit card fraud detection, sensor networks, fleet control etc. In such scenarios we want insight as fast as possible, from infinite flows of data, and multiple sources, using finite processing resources such as computational power and memory. There are also cases that may not have minimum latency as a necessary requirement, but the sizes of the datasets we handle are so enormous that force us to process the data in a streaming-like manner, because the we cannot afford algorithms with time complexity worse than $O(n)$.

2.4 Distributed Stream Processing

In modern applications, where having reliable and at the same time fast systems is crucial, the industry is quickly adopting the paradigm of distributed computing. A distributed system is a computing system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another, in order to achieve a common goal.

The advantages of using distributed systems over single computers, are numerous:

- Better Performance, i.e. greater amount of useful work accomplished, compared to the time and resources used

- Better Scalability, i.e. the ability to handle a growing amount of work in a capable manner or be enlarged to accommodate that growth.
- Higher Availability, i.e. The proportion of time a system is in a functioning condition.
- Fault Tolerance, .i.e The ability of a system to behave in a well-defined manner once faults occur
- Cost-effectiveness, as distributed systems use commodity hardware, which is a lot cheaper than supercomputers.

In our case, where we want to process *streams* of data, using distributed systems to do so has two extra benefits:

- Lower Latency, i.e. shorter period between collection of data and output,
- Higher Throughput, i.e. greater volume of data processed per unit of time.

However, not all algorithms can be implemented (at least efficiently) to run on distributed platforms and definitely not all can be used to process streams. In this thesis, we have chosen to implement and assess only algorithms that can run in a distributed way and process data on-line.

2.4.1 Comparison of relevant Frameworks

When building applications to run on distributed systems, it is preferable to use a suitable framework because it offers great abstraction taking care of all the consensus and coordination processes a distributed system needs to implement in order to be functional and reliable. By using such frameworks, a programmer only writes relatively high-level code, and the framework with its distributed runtime takes care of the rest.

The most popular, open-source software frameworks used for distributed stream processing are Apache Spark, Apache Storm, and Apache Flink. After a short presentation of each of them, we will select one to use and explain the reasons that led to this decision.

Apache Spark

Spark is a distributed general-purpose cluster-computing framework, that was initially created as a faster alternative to Hadoop MapReduce. Leveraging in-memory computation, instead of the disk-intensive computation that Hadoop uses, it is more "memory hungry" than Hadoop but also hundreds of times faster. This fact led to the quick adoption of Spark in the industry and growth of its popularity. The core data abstraction for Spark is the “RDD”, Resilient Distributed Data store. It has libraries for Graph Processing, Machine Learning and can process data in batch “and” on stream. However, it processes streaming data in micro-batches, and for this reason it cannot achieve very low latencies (near real-time).

Apache Storm

Distributed, (purely) stream processing computation framework. It uses custom created “spouts” and “bolts” to define information sources, and manipulations (also known as transformations) to allow distributed processing of streaming data. It was inspired from Hadoop MapReduce as well, and was initially seen as the streaming alternative to Hadoop, at a superficial level. It uses a similar general topology structure to a MapReduce job, with the main difference being that data is processed in real time. The downside of Storm is that it a bit old, and is not very functional without its library called “Trident”. Trident adds many features and abstractions that other computational engines offer by default, them being stateful stream processing, distributed querying of the data in the streams, joins, aggregations, grouping, functions, and filters. In addition to these, Trident adds primitives for doing stateful, incremental processing on top of any database or persistence store. Trident also has consistent, exactly-once delivery semantics [1].

Apache Flink

Flink is a distributed stream processing framework. While built mostly for streams, Flink’s runtime supports the execution of batch programs and iterative algorithms natively. It provides a high-throughput, low-latency streaming engine as well as out-of-the-box support for event-time processing and state management. Flink applications are fault-tolerant in the event of machine failure and support exactly-once delivery semantics. Moreover, it has libraries for graph processing, machine learning and a rich collection of data transformations and aggregations.

According to [16], who carried out a series of experiments and benchmarks to identify the strengths and weaknesses of each of these engines, Flink is the processing framework with the best performance in most cases, so we decided to use this one. The choice of framework though is not restrictive, as they all use very similar APIs and they all run on the Java Virtual Machine.

[1]We define these terms in the following pages

2.4.2 Apache Flink

In this section we will go through the features of Apache Flink [2], an overview of its distributed runtime and how it works under the hood, and also an overview of its programming model and APIs.

Zookeeper

First of all Flink in highly available setups depends on Apache Zookeeper, like other distributed engines of the Apache Foundation, to monitor the state and health of cluster. Zookeeper, which we will use and deploy in our cluster along with Flink, is a centralized service for distributed systems similar to a hierarchical key-value store, used to provide a shared distributed configuration service, synchronization service, and naming registry for large distributed systems. Zookeper can be viewed as an atomic broadcast system, through which updates are totally ordered. It is based on the Zookeeper Atomic Broadcast (ZAB) coordination protocol.

Programming Model

Concerning the Flink's programming model, Flink offers access to many layers of abstraction. The layers are from low to high:

- Stateful Stream Processing: The lowest level which contains Flink's basic abstractions and building blocks, the *stream*, the *state* and *time*. It allows users to freely process events from one or more streams, access the consistent fault tolerant state and register callbacks at event time or processing time (we discuss the Flink's *time* concept later).
- DataStream/DataSet APIs: Expressive core APIs for bounded/unbounded data streams and bounded data sets respectively. They offer lots of common building blocks for data processing like joins, aggregations, windows, state, etc. The DataSet API offers additional primitives on bounded data sets, like loops/iterations.
- Table API: Declarative DSL centered around *tables*, follows the relational model (schema, comparable operations etc). Goes through optimizer before execution.
- SQL: Closely interacts with the Table API to execute SQL queries.

For our implementations and experiments we will interact exclusively with the DataStream API, as it offers all the tools and expressiveness we need.

To write a program for Flink, one needs to be aware of its Dataflow Model. The Dataflow Model can be visualized as a DAG (directed acyclic graph), of which the edges are *streams* and the nodes are *transformations* on these steams. Each Dataflow DAG has one or more *source* nodes and one or more *sink* nodes, from which the data enter the computational graph and leave from respectively. Sources and sinks can be files from the local filesystem or persistent systems such as Amazon Kinesis, Apache Kafka, HDFS, Apache Cassandra, ElasticSearch and more, for which Flink provides ready-to-use connectors.

These Dataflows are inherently parallel. Each stream breaks in *stream partitions* according to the *parallelism* (number of *operator subtasks*) of corresponding operators. Streams can transport data between two operators in a *one-to-one* (or forwarding) pattern, or in a *redistributing pattern.* All this terminology is presented intuitively in figure 2.2 in page 25, which was compiled by images found in [2].

Distributed Runtime

Now we will see how the Dataflow model looks like in the distributed runtime of Flink. For distributed execution, Flink first optimizes the flow by chaining operator subtasks together into *tasks* if possible. A parallel instance of a task is called a *subtask*. *Each subtask is executed by one thread.* Chaining is important, because it reduces the overhead of thread-to-thread handover and buffering, and increases overall throughput while decreasing latency. In addition, execution is of course pipelined, meaning that operators start executing as soon as the previous ones have produced output, and obviously they don't wait for them to finish their execution on the entire stream first. In figure 2.2, the sample dataflow is executed with five parallel threads, because it has five subtasks.

These threads are executed in the Flink runtime, which consists of two types of Processes:

- The JobManagers (or *masters*), coordinators the distributed execution. A highly-available setup requires a quorum of JobManagers, on of which is the *leader* and the others are *standby.*

- The TaskManagers (or *workers*), who execute the subtasks in *task slots*, for which it is a good practice to set their total number equal to the number of CPU cores of the machine. A TaskManager is actually a JVM process. Tasks in the same process (thus in the same JVM) share TCP connections and may also share data, further reducing the per-task overhead and allowing for better resource utilization. Flink needs exactly as many task slots as the highest parallelism used in the job. If no parallelism is specified explicitly in a program, Flink will run it with the maximum value.

The distributed runtime can be seen in figure 2.3, taken from [7]. The *client* is the interface to the runtime, used to submit programs for execution and receive reports.

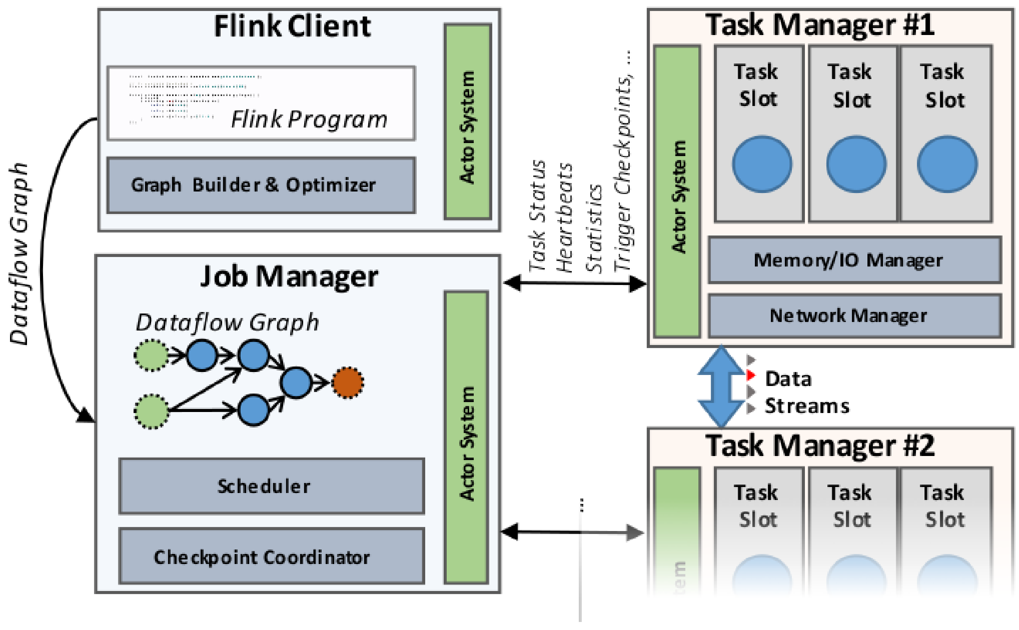


Figure 2.3: Flink's Distributed Runtime [2]

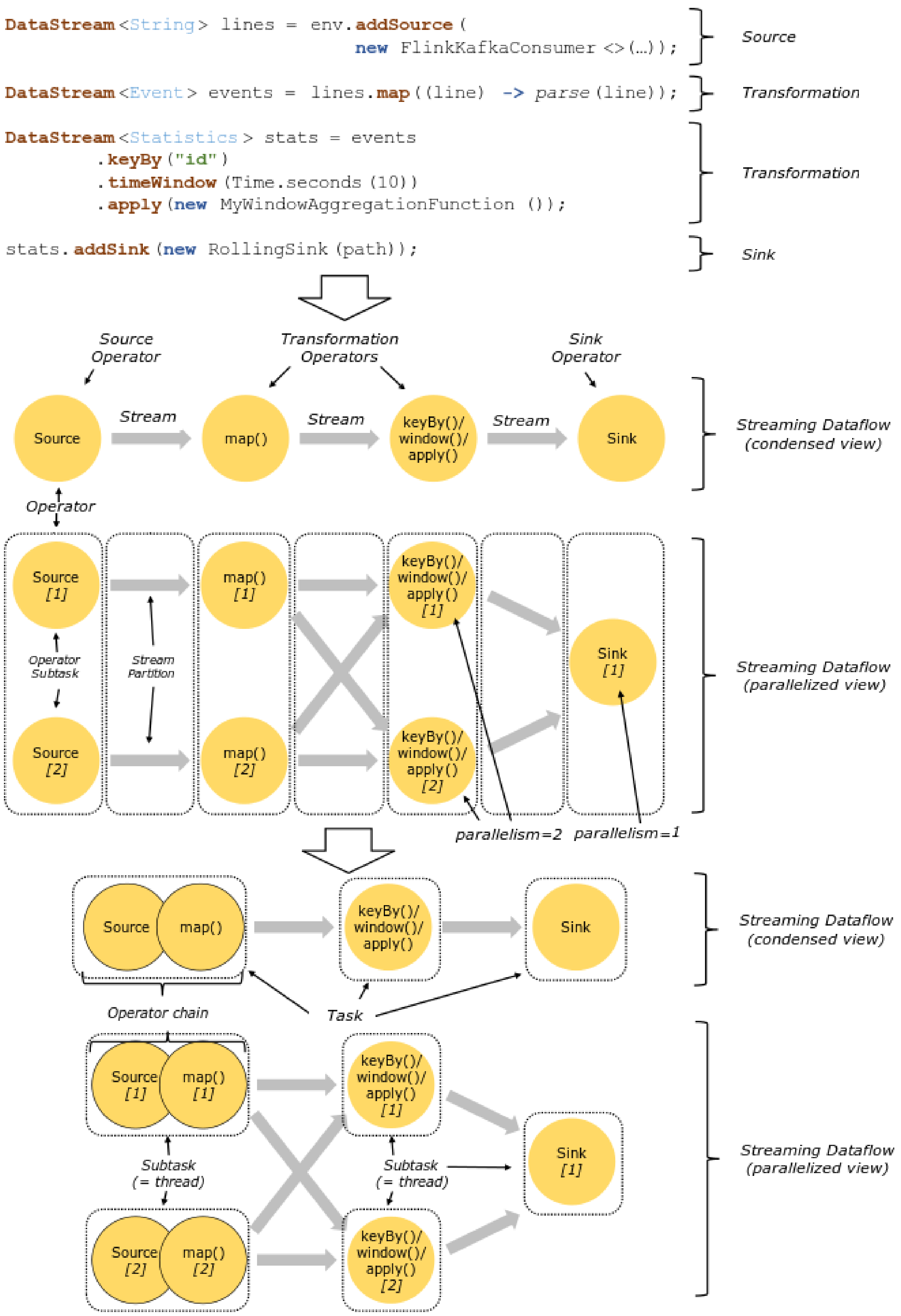
Flink can be deployed in a standalone cluster, which is the method we chose for the experiments in this thesis, or deployed using cluster resource managers such as YARN or Mesos.

Guarantees

- *Exactly-once* Delivery Semantics: Contrary to *at-most-once* semantics which can lead to lost messages and *at-least-once* semantics which can lead to duplicate processing, *exactly-once* semantics guarantee that each message will be processed exactly one time and no message is lost, even if nodes in the cluster die unexpectedly.

- Fault Tolerance: Flink implements fault tolerance using a combination of stream replay and checkpointing, which enable the system to recover from failed nodes.

Flink offers many more guarantees and abstractions, but they are out of the scope of the current work.



```
DataStream<String> lines = env.addSource(
                                    new FlinkKafkaConsumer<>(…));
DataStream<Event> events = lines.map((line) -> parse(line));
DataStream<Statistics> stats = events
            .keyBy("id")
            .timeWindow(Time.seconds(10))
            .apply(new MyWindowAggregationFunction());
stats.addSink(new RollingSink(path));
```

Figure 2.2: Flink's Dataflow Model [2]

Chapter 3

Implementation of Selected Algorithms

In this Chapter we present four algorithms that we chose for Anomaly Detection, describe their models, their characteristics, and we also provide some implementation details and define their parameters, which we will later use in the evaluation phase in Chapter 4.

We also provide all the possible distributed settings for each algorithm, for which we will use the following convention; "distributed" training or inference (or "on-cluster") is the respective action when executed, well, on the cluster, and "local" training or inference means "on a single computer".

3.1 Multivariate Gaussian Model

Anomaly Detection using a Multivariate Gaussian Model [21] is one of the simplest statistical-based anomaly detection techniques, yet it is often the most simple ones that are the most effective. It is based on the hypothesis that data are following the Gaussian (Normal) distribution. The idea is to create a model from the observations using this assumption, and apply it to newer observations to determine whether they are anomalies or normal.

Concretely, given an example $x \sim \mathcal{N}(\mu, \Sigma), x \in \mathbb{R}^d, x = [x_1, x_2, \dots, x_d]^T$, its probability density function is

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\Bigg(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\Bigg)[10],$$

where $\mu \in \mathbb{R}^d$ is the mean and $\Sigma \in \mathbb{R}^{d\times d}$ is the covariance matrix.

Now *assuming that variables $x_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$ are all independent*, we get:

$$p(x) = p(x_1, x_2, \dots, x_d) = p(x_1; \mu_1, \sigma_1^2)p(x_2; \mu_2, \sigma_2^2)\dots p(x_d; \mu_d, \sigma_d^2) = \prod_{j=1}^{d} p(x_j; \mu_j, \sigma_j^2),$$

where:

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\Big(- \frac{(x-\mu)^2}{2\sigma^2}\Big)$$

To train the model ([21]) (which actually consists of the values $\mu_i, \sigma_i^2, \forall i \in [1, \dots, d]$), one needs to:

- Calculate $\mu_i = \frac{1}{n}\sum_{j=1}^{n} x_i^{(j)}$, where n is the number of training examples (the size of the training dataset),
- Calculate $\sigma_i^2 = \frac{1}{n}\sum_{j=1}^{n} (x_i^{(j)} - \mu_i)^2$

Then, in the evaluation phase, given a new example x, we compute:

$$p(x) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi}\sigma_i} \exp\Big(- \frac{(x_i-\mu_i)^2}{2\sigma_i^2}\Big),$$

and we flag x as anomaly is the value of $p(x)$ is smaller than a threshold value ϵ (hyperparameter).

3.1.1 Features

The assumption that all dataset features are independent is of course a strong hypothesis, but it leads to a model which offers great computational benefits, .i.e. it scales much better in large datasets, plus it can be easily implemented with functional operators, which are present in the APIs of all modern Big Data frameworks. It is also applicable to datasets with a large number of features and/or few training examples, whereas for the full Multivariate Gaussian model, the number of training examples *must* be greater than the number of features, otherwise the covariance matrix Σ is non-invertible and therefore we cannot use the model.

Also, we can reduce whatever negative effects this decision may have, simply by identifying (computationally or even intuitively), which features are related to each other, and add to our datasets extra features to express this correlation. For example, in a dataset in which we know that feature x_1 grows in value “together” with feature x_2, we can add the values $\frac{x_1}{x_2}$ as an extra feature and the algorithm will perform as expected.

To train the model, a relatively small number of unlabelled data is needed. Then both training and inference can be performed on batch data, or on-line.

3.1.2 Implementation

This algorithm can be implemented in a Big Data framework in a very straightforward way, if the functional operators *map* and *reduce* are available. Assume that our training data are n d-dimensional vectors $x^{(j)} = [x_1^{(j)}, x_2^{(j)}, \ldots, x_d^{(j)}]^T, j \in [1, \ldots, n]$. Then the values of vectors μ and σ^2 can be computed by:

```
// Scala
dataSet
  .map { x => (x, pow(x, 2), 1) }
  .reduce { (x1, x2) => (x1._1 + x2._1, x1._2 + x2._2, x1._3 + x2._3) }
  .collect()
```

This way, by having the tuple $T = \left[\sum_{j=1}^{n} x^{(j)},\ \sum_{j=1}^{n} \left(x^{(j)}\right)^2,\ n\right]$ available at any point of the computation (when training or evaluating, in batch or on stream), we have access to all parameters of the model μ and σ^2 at any time by calculating:

$$\mu = \frac{\sum_{j=1}^{n} x^{(j)}}{n} = \frac{T[0]}{T[2]}, \qquad \sigma^2 = \frac{\sum_{j=1}^{n} \left(x^{(j)}\right)^2}{n} - \mu^2 = \frac{T[1]}{T[2]} - (\frac{T[0]}{T[2]})^2$$

We have chosen to implement this algorithm with batch training and of course on-line deployment (because of some particular characteristics of the framework used). We also added a function that suggests automatically a good threshold value to use, by using a labelled dataset and finding the value ϵ that maximizes the F1 score of the results.

Parameters

This algorithm does not have any parameters of its own. We only added a hyperparameter which we will call `Iter` (integer), which will be the number of iterations it will do over a labelled evaluation dataset, in order to find a good threshold value. By default it has a value of 100.

Distributed Settings

This algorithm's both training and inference routrines can be programmed easily to run in a distributed way. One option is to train on-cluster (or locally for smaller datasets) and then employ the same model in all parallel instances (mappers) for on-line inference, and the other option is to not use training at all, and make the algorithm run purely on-line. In the latter each parallel instance will build its own model.

In our experiments we implemented on-cluster training and then on-cluster inference[1].

3.2 Isolation Forest

Isolation Forest [19] utilises the concept of *isolation* to detect anomalies in the dataset. It takes advantage of two quantitative properties that anomalies have:

- Anomalies are the minority, consisting of fewer instances, and
- They have feature values which are very different from those of normal instances.

These two characteristics make anomalies susceptible to *isolation*, meaning that they are more likely to be isolated from other instances when the dataset is randomly partitioned.

This algorithm works by recursively randomly partitioning the dataset until it reaches a particular depth or isolates a point. To represent the partitions, it uses a special kind of binary search tree (BST) called *iTree*. The idea is that *anomalies, since they lay further from the rest observations, will require a lower number of random dataset partitions to become isolated, whereas normal observations, will need a higher number, as they are close to other normal points.* This translates to respectively *shorter and longer path lengths (or distances from the root node)* in each *iTree*. The anomaly score that is inferred for an example during the evaluation stage is based on this *path length* value. For example, in fig. 3.1, we can see that point x_o requires on average fewer "cuts" to be isolated, than point x_i.

The model of Isolation Forest is composed of an ensemble of *iTrees*. Each *iTree* is built on a subsample of the original dataset. The use of subsamples has some very useful properties (see section 3.2.1). Each subsample is formed by randomly picking instances from the whole dataset without replacement.

The anomaly score for a data point is the average value of the path lengths acquired by "passing" the data point from each *iTree*.

3.2.1 Features

Most model-based anomaly detection algorithms construct a profile of normal observations, and then identify as outliers the instances that do not conform to this normal profile, like the Gaussian-based algorithm we described earlier. However,

[1]The on-cluster training has only one layer of "reducers", which creates a bottleneck. When used in enormous datasets one may want to use multiple layers of reducers.

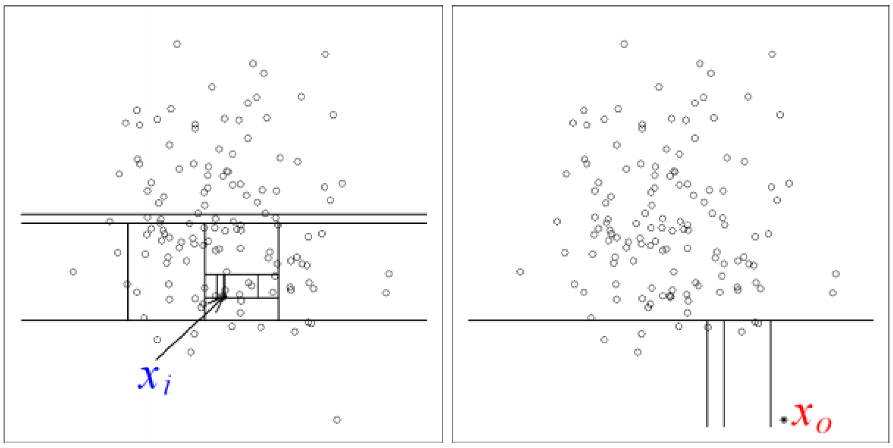


Figure 3.1: Example of data partitioned by the Isolation Forest algorithm [19].

this approach often causes a high computational complexity for data of higher dimensionality, and, because the model is optimized to profile normal points, and not to detect anomalies, it often ends up with too many false positives, or few true positives (not to mention that most of the times a labelled dataset is mandatory for the training phase). Isolation Forest is different from such algorithms because its model isolates anomalous instances instead of profiling the normal ones, requires no labelled dataset (it is an unsupervised algorithm) and is also robust when applied at data with high dimensionality.

Also, most distance-based and density-based methods do not handle the effects of *swamping* and *masking* [20] well, having poor performance in such cases. *Swamping* is the when normal instances are too close to anomalies, causing them to get incorrectly flagged, and *masking* is the situation in which too many similar anomalies form a small cluster, concealing their presence. Isolation Forest alleviates the effects of these two situations by operating on random subsamples of the original dataset.

Another great feature of Isolation Forest is that it has a linear time complexity. Specifically, a complexity of $O(t\psi \log \psi)$ in the training phase and $O(nt \log \psi)$ in the evaluation stage, where ψ is the subsampling size (256 is the recommended value), t is the number of iTrees in the forest ensemble (around 100 is a good choice) and n the size of the dataset. In addition, it has a relatively small memory footprint because of the subsampling, making it able to handle extremely large and high-dimensional datasets without requiring much memory.

3.2.2 Implementation

Despite being a great algorithm, it has a downside. Isolation Forest is not efficiently parallelizable for clusters of machines. This is due to a number of reasons:

- During the training phase, each subsample (from which the corresponding *iTree* is built), has to be created by randomly picking instances from the whole dataset, without replacement, and all the subsamples must be of equal size. In the case that the dataset is stored centrally, the source node in the DAG of computation has to do all the subsampling and then pass the subsamples to the worker nodes so that each parallel slot can build one or more *iTrees*. The subsampling at the source node creates a bottleneck. In the case that the dataset is stored in a distributed way (e.g. in HDFS), the creation of equally-sized subsamples is painfully difficult and inefficient because each source node that loads some blocks from the distributed storage has to be aware of how many instances each other source node has picked for the formation of the same subsample. A naive approach to this would be, instead of picking random instances to form a subsample, to use the whole block from the distributed storage as a subsample for training. This is wrong, as the subsamples are not random anymore and the trees become "biased", leading to incorrect results.

- During the inference phase, if each tree "lives" in a different node, each example from the input stream, in order to be evaluated, has to be broadcast to all worker nodes, and then the result of each of them has to aggregated with the rest, so that the *average path length* (and thus the anomaly score) can be calculated, which again, creates a bottleneck, leading to limited scalability.

Then one might wonder, "why even chose the algorithm then?". Well, it has a remarkable characteristic; *Contrary to existing methods where large sampling size is more desirable, isolation method works best when the sampling size is kept small [19].* Large sampling size reduces iForest's ability to isolate anomalies, as normal instances can interfere with the isolation process and therefore reduces its ability to clearly isolate anomalies. The anomaly detection capabilities also cap at a subsample size of around $\psi = 256$, and thus, there is no need to increase it further. In addition, as it shown in [19], the model converges quickly even with a small ensemble size, so there is no need to increase the number of trees too much (fig. 3.2).

These qualities (small subsample size, few trees) allow us to *train the model centrally*, in a single machine, and then export the trained model. The saved model is then loaded by all parallel instances of the cluster, and then the evaluation stream can be partitioned, achieving *on-line inference* with maximum parallelism and maximum throughput.

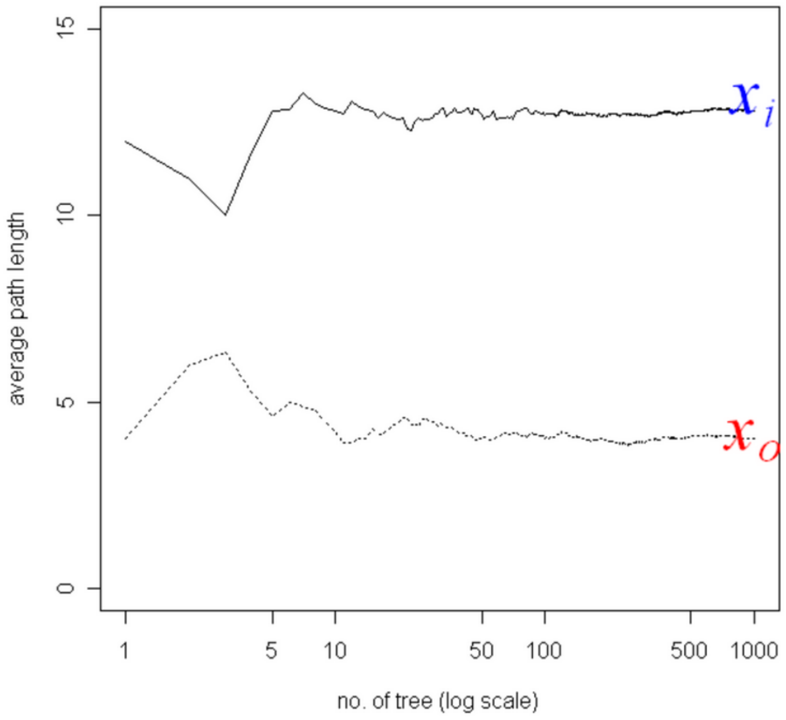


Figure 3.2: Average path converges quickly [19].

One advantage of training centrally, is that there is no network overhead in the training procedure. Network I/O would cause a big delay, to process rather small subsamples (a job that does not require a whole node to be involved). Obviously the tradeoff is not worth it. In addition, while training centrally, since the trees are independent from each other, their creation can still be done in a parallel manner (if the machine has more than one core).

They way we serialize the model is by traversing in a *preorder* fashion each tree, forming an array of nodes, ordered by the way they were visited. Then we put these arrays in a meta-array, and export this as a JSON. Deserialization then follows the same process in reverse.

Parameters

Our implementations of IForest have a number of parameters:

- `Auto` (boolean): If set to false, contamination mode is used.
- `Contamination` (float): Contamination is a value between 0 and 1, indicating the expected percentage of anomalies. If we know for example that in our dataset, 5% of the observations are anomalies, we can use this fact to yield better predictions. By default IForest flags every point with score > 0.5. In Contamination mode, it will flag the top n scores, where n is the number of

anomalies calculated with the contamination parameter.

- `NumTrees` (integer): The number of trees present in the ensemble
- `SubsampleSize` (integer): The subsample size

Distributed Settings
This algorithm, as we clarified earlier, cannot be efficiently trained on-cluster, but it also doesn't need to. The model is trained centrally (using multiple threads for the trees is possible), then the created model is exported, and loaded by each parallel instance of the cluster. Then on-line inference is possible efficiently on-cluster.

3.3 Half-Space Trees

This algorithm [25] is quite similar to the Isolation Forest, in the sense that its model is an ensemble of trees too. However, it uses a different approach to rank anomalies, a concept called "mass" [26], and, contrary to the Isolation Forest, it can be trained *and* deployed on-stream, as it is mainly an online algorithm.

The trees that this algorithm uses closely resemble full binary search trees (well-balanced BSTs). Each internal (non-leaf) node represents a random partitioning of the data. More specifically, HS-Trees selects a random feature every time and cuts the data recursively, right at the mid-point of the range of the selected feature.

Now the training and the inference is done using two windows, namely a "reference" and a "latest" window. Every new example that is fed to the algorithm by the data stream, is used to make a prediction (to yield an anomaly score), using the model built in the *reference* window, and is also used to update the model by updating the model in the *latest* window. Obviously when the *latest* window is full, the *reference* window is discarded, the *latest* becomes the reference, and a new *latest* is created.

Each arriving example goes through each node of the model in the latest window, by following the comparisons made with the corresponding splitting feature and value. For example, if the 2-featured example $[0.1, 0.3]$ goes through a node with splitting feature 2, at value 0.5, the feature will follow the left node, because $0.3 < 0.5$. This continues recursively until a leaf-node is reached. Each leaf-node keeps counter that is incremented as soon as an example "falls" into this leaf (we can think of them as buckets). The value of this counter is referred to as *mass*. As it is evident, *data points which fall into leaf-nodes with low mass, are likely to be anomalies, since a low mass profile is caused by a "sparse" area of the data space.* This behaviour is demonstrated in figure 3.3.

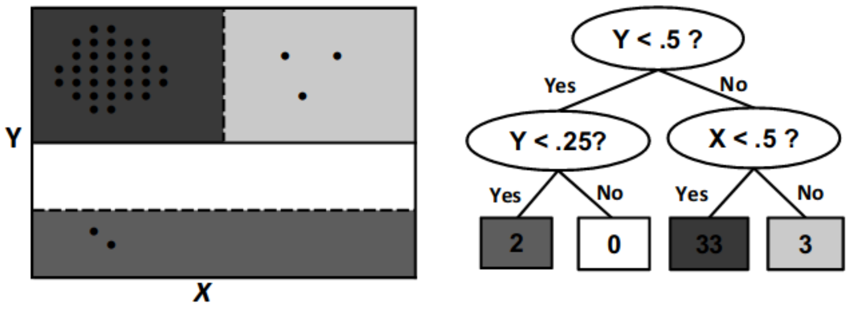


Figure 3.3: Example of data partitioned by a simple HS-Tree [25]

Afterwards, the example goes through the *reference* window, so that an anomaly score can be calculated. With the same procedure, we find out in which leaf-node the example belongs, and we use the mass in that leaf-node to calculate the anomaly score. This is done for every tree in the ensemble, and the final anomaly score is calculated as the total sum of these scores.

3.3.1 Features

HS-Trees is a powerful algorithm, as it achieves high detection rates while using constant memory to process infinite data streams. It has an amortized time complexity of $O(t(h+\psi))$ [25], where t, h, ψ are all small-valued hyperparameters, and of course there is no dependence on the size of the input.

It requires no previous knowledge of the dataset, although, to bootstrap the algorithm properly, an approximate range of each feature needs to be provided, in order to estimate the range in which the splits need to belong. Also, because its performance is contingent on the diversity of each randomly created tree, these ranges need to be a bit randomized before the creation of each tree. The original paper [25] though, recommends that the data are normalized in the range $[0, 1]$.

One key characteristic of HS-Trees, is that it has the ability to adapt to *evolving streams*. Some data streams may change slowly over time, causing a "shift" of the normal observations. Due to the windowed nature of this algorithm, it is able to keep its model up-to-date and not suffer from *concept drift*, which can cause other algorithms to flag everything as outliers.

If the algorithm is used on evolving streams, it is recommended that it doesn't update the model after the end of every window, but after a fixed number of *changes*, where a change is defined as a difference in the number of anomalies found during a window session. For example, if we are identifying 3 anomalies on every window, and they suddenly become 0 for one window, it is not recommended to update the model immediately, but instead increase a counter of changes by 1 and only update if the behaviour continues in the upcoming windows.

Concerning the randomizations, as with Isolation Forest and other ensemble based algorithms, the negative effects of using random number generation is cancelled out by aggregations (e.g. summations to calculate the mean).

3.3.2 Implementation

The algorithm is not hard to implement, especially if recursion is used, as described in the original paper [25]. However, the recursive implementation is slightly slower because of the amount of function calls. An implementation with iterations over an array that represents the full binary tree (with node i having its children at positions $2i+1, 2i+2$), is somewhat faster (array-structured data often make better use of the CPU caches too, leading to faster executions).

Also, there is no need for two windows. One window can be used (basically a counter), with each leaf-node having two variables, namely r and l, corresponding to the values that each window would have. We infer scores based on the r value, and at the same time update the l value.

Parameters

The parameters of HSTrees in our implementations are the following:

- `NumTrees` (integer): Number of trees in the ensemble.
- `SizeLimit` (integer): Optimization parameter used to stop access to nodes with too small mass.
- `MaxDepth` (integer): Maximum depth of each HSTree
- `WindowSize` (integer): Size of the window

We also used a hyperparameter to help with the estimation of a good threshold value. Because this algorithm is purely on-line, we had to come up with a way to estimate a good threshold automatically so that we could perform the experiments with the other algorithms, some of which don't even require always a threshold (IForest). This hyperparameter is called `EstNumAnomalies` and is used in Chapter 4.

Distributed Settings

With HSTrees being purely on-line, the best option here is to let every parallel instance build its own ensemble of HSTrees, and this is what we did. Another option would be to have only one universal ensemble, a central model, and infer all the streams from this but this of course is not scalable.

3.4 LODA (Lightweight On-line Detector of Anomalies)

LODA [22], as the name suggests, is a lightweight, model-based, ensemble-based detector that can operate to identify deviating samples on both batch and streaming data.

The model of LODA is comprised of a collection of k one-dimensional histograms (denoted $h_i, i = 1, ..., k$), each approximating the probability density of input data. Input data (e.g. $x \in \mathbb{R}^d$) is projected into each histogram via k projection vectors, denoted $w_i \in \mathbb{R}^d, i = 1, ..., k$ [22]. The output of the algorithm, the anomaly score for data point x, is proportional to the negative log-likelihood of the sample, and given by:

$$score(x) = -\frac{1}{k}\sum_{i=1}^{k} \log \hat{p}_i(x^T w_i)$$

The closest the score is to zero, the more likely it is that x is an anomalous example. The sparse projection vectors w_i are the means to diversify individual histograms, and are created at the algorithm initialization, by generating a set of k d-dimensional vectors with $\sqrt{d}$ non-zero elements drawn from $\mathcal{N}(0, 1)$.

Now the histograms can be of three types, namely fixed-bin, equi-width and on-line histograms [23]. Each of these options has different properties and drawbacks which are presented in table 3.1.

3.4.1 Features

LODA, like HS-Trees, can keep its model updated when deployed on-line without suffering from *concept drift*. LODA however does not require any preliminary operations on the data, (unlike HS-Trees, which requires normalization in the range $[0, 1]$ of each example), and only requires features to have approximately the same scale [22], which is a more relaxed condition.

LODA is also robust against *data with missing values* meaning that it delivers satisfactory results even when some variables are absent. This is done by calculating the output only from those histograms whose projection vector has a zero one the place of missing variables [22].

The time complexity of LODA is $O(kd)$ and the space requirements are constant (actually when using the equi-width histograms they can in theory grow indefinitely, but this can be prevented by either discarding bins that have not been updated for a long time (after a concept drift for example), or by redefining the widths of the bins and some sort of merging).

Histogram type	Advantages	Disadvantages
Fixed-Bin	Constant space requirements and $O(1)$ time for updates and access to the probability estimate value	For a good choice of number of histogram bins it is required that the range of the data is at least approximately known beforehand
Equi-Width	$O(1)$ time for updates and access to the probability estimate value, does not require knowledge of the range of the data	Required space is not restricted and can theoretically grow infinitely
On-Line	Constant space requirements, does not require any knowledge of the data range or distribution beforehand	Not as fast as the fixed-bin or equi-width histogram

Table 3.1: Comparison of types of histograms for LODA

3.4.2 Implementation

In [22], it is shown that the equi-width histogram delivers by far the best results, and is also the best choice for on-line deployment. If some sample data are available, a value for the bin-width can be easily determined. The data structures we used to represent the histograms are simple hash-maps with keys $k = \lfloor \frac{x}{\Delta} \rfloor$, where x is the value to be inserted.

Parameters
LODA has the following parameters in our implementations:

- `NumFeats` (integer): Must equal the number of features of the dataset
- `BucketWidth` (float): The width of the bins (or buckets)
- `NumVectors` (integer): Number of projection vectors in the ensemble

Two extra hyperparameters which we had to add like in previous algorithms: `TrainSize` (integer), which is the number of instances in the beginning of the stream that should update the model but not be used to infer anomaly score, since the model will not be ready yet, and `ProbSumBorder` (float), the threshold.

Distributed Settings

LODA, thanks to its similarities with HSTrees, has the same exact distributed settings.

Chapter 4

Experiments

4.1 Infrastructure

For the following experiments, we used two types of infrastructure, a personal laptop and an ensemble of Virtual Private Servers which formed the actual computation cluster we used to run Apache Flink and the distributed environment on.

The experiments that were conducted in order to compare the detection capabilities and efficiency of the algorithms, both on synthetic datasets and real ones were done on the personal machine, using implementations in Python v3.6 (based on the NumPy scientific computing library).

The other experiments we did to study how the algorithms can perform at scale, when deployed in a distributed environment on top of a distributed stream processing framework, were of course conducted on the cluster, using implementations in Scala v2.11 and the APIs of Apache Flink v1.6.1.

This cluster comprised of 4 *"worker" nodes*, and 1 "master" node, each one being a VPS, a Virtual Machine essentially, created on the OpenStack platform. Due to the abstraction that OpenStack offers, we cannot be sure if the VMs were hosted on the same physical machine or not, but this is of minor importance as small differences in the latency will fade out as the cluster becomes bigger.

Each node has 2 vCPUs (2 cores of a physical CPU), and 4GB of RAM, and 40GB of storage space (HDDs), running Ubuntu 16.04. These resources may seem quite limited but they are more than enough for the JVM to run the `JobManager` and `TaskManager` processes of the Flink runtime.

With these settings, we can achieve a maximum parallelism value of $2 * 4 = 8$. This should be enough to determine the throughput of the system in relation to the amount of parallelism.

4.1.1 Computation Cluster Automation

When dealing with distributed platforms, i.e. when deploying systems across multiple machines, one can immediately understand the difficulties that emerge when it comes to setting the correct system environment and framework configurations. To face these problems, we decided to use an open-source provisioning, deployment and configuration management tool, Ansible [1].

Ansible is incredibly useful, because with the use of simple *modules* that it provides, declarative YAML files, variables and templating, one can fully automate the whole process of application deployment and configuration, and restore every system's state in the case of an event that causes changes.

The main reason we selected Ansible over other provisioning tools, is the ease of use and the fact that the only requirement for the target machines is an SSH connection. Other tools usually require the installation of client software, but with Ansible, it is possible to create a VM from OpenStack and have it directly and automatically install all the packages needed and run our services, without even logging in to the machine.

Ansible works by checking the system state we define via some special instructions, and restoring this state when needed (idempotency). For example, if we need a package installed in a group of nodes, we will use the Ansible module that corresponds to the package manager of our target OS, and then declare that this package should be in the state *present*. Then, every time we run Ansible, it will log via SSH into every node belonging in this group (defined in an *Ansible hostsfile*) and make sure that the package is indeed installed. If it is not, it will install it and inform us about it at the end of the run.

In our use case, our *Ansible Playbook* as it is named, firstly downloads, configures, installs and runs Apache Zookeeper, a dependency of Flink, as a service (with `systemd`) and then does the same steps for Apache Flink itself. But besides the obvious benefits of such great automation, less perplexity and complexity, easier management, time efficiency, smaller probability of making mistakes etc., there is also something we gain which is far more important: *Abstraction*.

The cluster setup is not longer dependent on which and how many nodes we want in our ensemble. By editing the *Ansible hostsfile*, we can specify directly the hostnames (or IP addresses) of the nodes we want to belong in the group that runs for instance Zookeeper, and in the group of nodes running Flink, and then run Ansible, which will take care of the rest.

More specifically, Ansible will firstly log into every node and gather information from each machine, and then store that information in a JSON object called `hostvars`. Then it will go ahead and use the templates that we have written and by using these along with the information collected and some variables we have specified, it will generate all the necessary configuration files, check if the corresponding ones in the VPSs are different, and update them if they are. For example, in our setup, the configuration files of Apache Flink have a property called `maxParallelism`, which has to be set manually. With Ansible and `hostvars`, we find out how many vCPUs we have in the cluster, sum them up, and set this parameter automatically. So even if we add a new node with a four core CPU, ansible will update this parameter correctly in the configuration files.

Also, in case that something goes wrong with the VMs or say, for industrial deployments, we want to change cloud provider, this change is a matter of a few Ansible commands.

4.2 Datasets

To assess the performance and gain some insight on the behaviour of each algorithm in relation to the type of data it is fed with, we used both artificially created datasets (synthetic) and real ones.

For the synthetic ones, we created datasets formed by *clusters* of data points and other points scattered around these clusters. The rationale for this decision comes from the definition of anomalies and normal observations; Since normal observations are the norm, they lay close to each other usually creating dense areas, which we will refer to as “blobs”, while novelties fall outside of these areas.

The number of clusters present in our dataset is a characteristic we later examine, and we will refer to it with the term *clusterization*.

These clusters are each comprised by data points following a Gaussian distribution. So on the one hand we generate the normal observations (the inliers) to form these clusters, and then generate the outliers from the uniform distribution (representing the noise), randomly scattered across an area slightly larger than the area the blobs cover (possibly included in the blobs too).

The outliers make up a percentage of 5% of all observations, a quite realistic scenario. The total number of observations, along with the total number of features and the clusterization number are “variables” for our performance measurements.

For a real world dataset, we used the KDD Cup 99 dataset from the UCI machine learning repository, which is well-known in the field of Anomaly Detection, Classification, and especially Network Intrusion Detection, and used very often in relevant benchmarks.

We chose this dataset because it is labelled, it contains about 4.9 million examples, and it also has a high number of features, 41, out of which 7 are categorical and 34 continuous.

For the accuracy and efficiency experiments running in a personal computer, we did not use the whole dataset. We used a reduced version (10% of the original) and only the HTTP and SMTP components. More specifically, we created a dataset from the observations of SMTP observations with a size of 9723, 125 of which (1.2%) are observations connected to network attacks (anomalies). We removed 3 non-numerical features (strings), which leaves us with 38 features, and used 0-1 as labels instead of the original labels (strings describing the type of the corresponding attack). We will refer to this dataset as the SMTP dataset. The second dataset created from the HTTP rows, has 64293 rows with 2407 being anomalies (3.7%). In this HTTP dataset we did not include many features, and only chose the 3 most important for the kinds of attacks relevant to HTTP according to [17], and also the most accessible (in a real world application) which are the connection duration, the number of source bytes and the number of destination bytes.

Finally, for the measuring of throughput in the distributed environment, we used the full KDD dataset (4,940,210 examples) as we needed a good amount of data to stress the distributed setup for the experiments.

4.3 Measurements

4.3.1 Evaluation Metrics

In the field of Machine Learning, especially when the application is *Classification*, which is similar in a sense to *Anomaly Detection*, the most simple metric used to rank the performance of the algorithm is the ratio of the number of correct predictions, to the total number of predictions. However, this approach is not very suitable for Anomaly Detection applications, because in such cases, the number of outliers is usually very small in relation to the size of the dataset (about 1-5%).

A simple example to understand the inapplicability of this metric for our cause would be the following; Suppose we have an algorithm that classifies our observations correctly 95% of the time. In most classification applications this is considered a good accuracy score, but in the context of Anomaly Detection, where we can have for example a dataset with 5% of the total points being outliers, this algorithm may completely fail to flag any anomaly as such, and still deliver a 95% classification accuracy.

Thus, we will have to use a different metric. Popular options include [11] *Logarithmic Loss*, which works well for Multi-Class Classification (which is not our case), *Mean Absolute Error* and *Mean Squared Error*, good for Regression (again, not our case), and metrics based on the numbers of *True Positives* (TP), *True Negatives* (TN), *False Positives* (FP), and *False Negatives* (FN), such as the F1-Score (or balanced F-Score) and AUC (Area Under Curve) methods. Generally, F-Scores and AUC scores are good for Anomaly Detection algorithm evaluation because they penalise mispredictions based on their type, and not only on their numbers.

To define the metrics, we will to define first 4 key ratios:

- $Sensitivity = \frac{TP}{TP+FN}$
- $Specificity = \frac{TN}{TN+FP}$
- $Precision = \frac{TP}{TP+FP}$
- $Recall = \frac{TP}{TP+FN}$ (same with Sensitivity)

Intuitively, expressed with the terms of Anomaly Detection, Sensitivity (or Recall) measures the proportion of actual anomalies that are correctly identified as such, Specificity measures the proportion of actual normal observations that are correctly identified as such, and Precision measures the ratio of correct classifications as anomalies (positives) to the number of all observations labelled as anomalies by the detector.

The choice of what measure to use depends on the type of the application. For example in applications such as fraud detection, where it is crucial to not mislabel an actual fraudulent transaction as normal, we want to have a model with high Sensitivity (or Recall), whereas in applications where the cost of having a false positive is high, we want to increase the model's Specificity and Precision. For example, in email spam detection, labelling a non-spam email as spam has a greater cost than the opposite.

Now based on these ratios, we have:

- *F1 Score* $= 2\frac{Precision*Recall}{Precision+Recall}$, the harmonic mean of precision and recall, and
- *AUC Score*, which is of two types:
  - *AUC ROC, or AUROC* which stands for Area Under Receiver Operating Characteristic curve, and
  - *AUC PR*, for Area Under Precision Recall curve.

The **F1 score** keeps a balance between Precision and Recall, giving equal importance to both. This is why it is used quite often as a good ranking method for Anomaly Detection algorithms. However, as we emphasized before, evaluation should be consistent with the actual application and requirements. In the current work, since we are performing a more generalized evaluation and comparison, it is not improper to use the F1 score as a metric.

Another fact that favours the selection of this metric is that it can be used to evaluate an Anomaly Detection system in an on-line manner.

The downside with the F1 score however is that it is dependent on the threshold value selected for classification (which is basically a hyperparameter and requires tuning for every dataset the algorithm is applied onto). The absense of a good threshold value leads sometimes to poor F1-Scores, failing to provide a more hollistic view of an algorithm's effectiveness.

This problem is solved with the use of the AUC metrics. These do not depend on the setting of the threshold hyperparameter, as they average over all possible thresholds. It has two types: AUC ROC and AUC PR. The AUC ROC (or AUROC) metric is the area under the ROC curve [27], which is the curve formed by plotting the Sensitivity (or True Positive Rate - TPR) against the False Positive Rate (FPR), defined as $\frac{FP}{FP+TN}$. The AUC PR is the area under the curve formed by plotting Precision against Recall.

Between these two metrics, we chose the first, **AUROC**, because AUC PR is, according to [9] best for skewed datasets, i.e. datasets suffering from the *class imbalance problem.* The class imbalance problem appears when the total number of a class of data (positives in our case) is far less than the total number of another class of data (negatives). Since the datasets we have used, both synthetic and real have a decent percentage of anomalies (around 5%), there is no need to use the AUC PR metric, which is more "limiting" in the sense that it does not account for true negatives at all.

Finally, as a measure of the performance in the cluster we used the **throughput**, as it is commonly used in such cases. The throughput of a system is defined as the rate at which the system can process data.

4.3.2 Detection Accuracy

The first characteristic we are going to examine is the number of "clusters" present in the dataset.

Each dataset used consists of 1000 data points, 5% of which are outliers. For the number of features, we created datasets with 2 features because with that setting, it is possible to create scatterplots in 2D and visualize in a very intuitive way the performance and behaviour of each algorithm.

The settings for each algorithm are shown in table 4.1. The explanation of each of these settings can be found in Chapter 3.

For the Multivariate Gaussian algorithm, judging by the scatterplots in figure 4.2,

Algorithm	Settings	Algorithm	Settings
Gauss	Iter: 1000	IForest	Auto: True
			NumTrees: 100
			SubsampleSize: 256
HSTrees	NumTrees: 50	LODA	NumFeats: 2
	SizeLimit: 25		BucketWidth: 1
	MaxDepth: 5		NumVectors: 100
	WindowSize: 256		TrainSize: 500
	EstNumAnomalies: 50		ProbSumBorder: -4.1

Table 4.1: Parameter settings for the Clusterization Experiments

we can see that it is very effective for the dataset containing only 1 cluster, which is expected, since the algorithm's model is created from the exact same distribution from which the inliers are drawn. However, we can see that as the clusterization number increases, the algorithm fails to flag some outliers as anomalies, in particular those that lie in between the blobs. This is happening because in order to determine if a data point is an anomaly, a probability threshold is used, as described in Chapter 3, and this threshold corresponds to the border of an area in the contour plot of the Gaussian Model, outside of which everything is flagged as an outlier. This can be visualized by the red curve in figure 4.1.

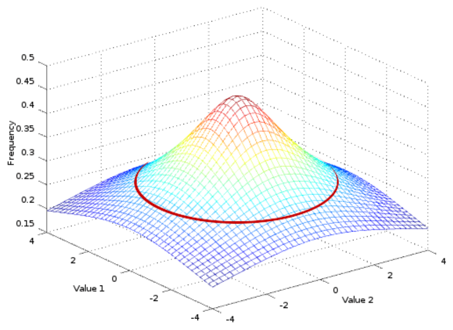
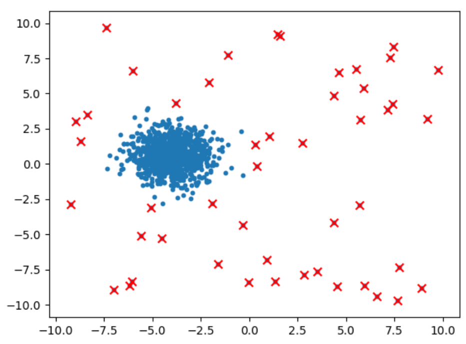
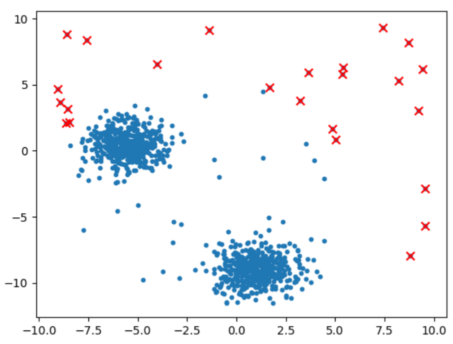


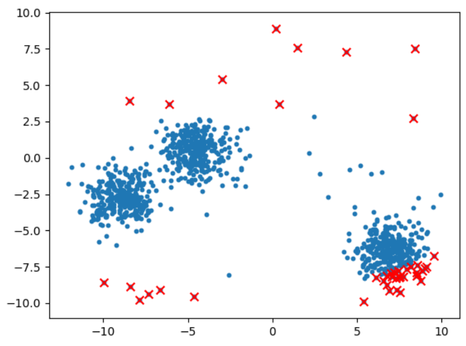
Figure 4.1: Visualization of the probability threshold of the Gaussian Model

Obviously, the observations that lie between two or more blobs, will have a probability value greater than the threshold and therefore cannot be flagged. This is a weakness of the Multivariate Gaussian algorithm, which may limit its applicability to some datasets.

For the Isolation Forest performance in relation to the number of clusters in fig. 4.3, we can see that there may be a connection between the number of clusters in the data and the "aggressiveness" of the algorithm towards the outliers. For


(a) 1 cluster


(b) 2 clusters


(c) 3 clusters

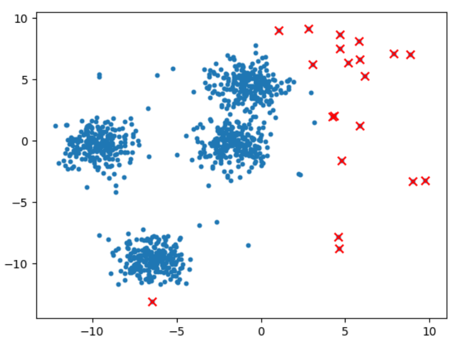

(d) 4 clusters

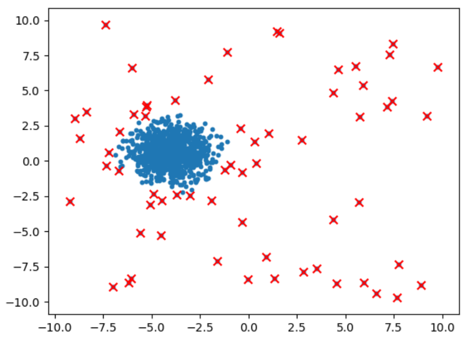
Figure 4.2: Multivariate Gaussian performance vs. Clusterization

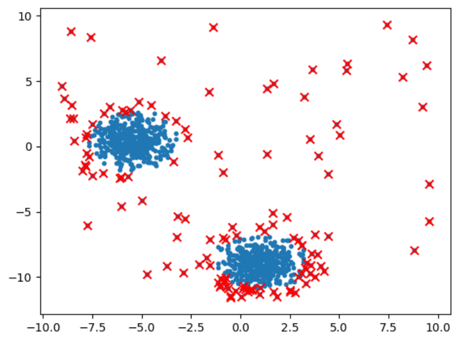
instance, when having 4 clusters, the algorithm is labelling too many negatives as positives (we have a high number of False Negatives).

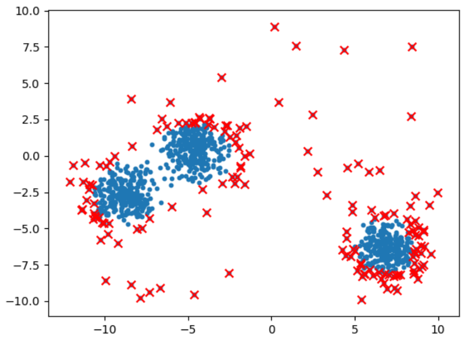
Next, we have the Half-Space Trees in figure 4.4. Despite being an online algorithm, it does a good job detecting anomalies in general, though it takes some time to bootstrap the model (a certain number of instances equal to the `WindowSize` parameter to be exact).

We observe however that, it misses some outliers, those that lie in lane-like areas extending from a blob vertically and horizontally, especially those that lie in the crossing of two such areas (from two different blobs). This weakness probably has to do with the fact that the splits of the plane that the algorithm uses to profile the mass of an area are vertical to the axes.

The phenomenon is observed less frequently when the number of the features of the dataset is high, or when the number of HS Trees used in the ensemble is high.


(a) 1 cluster


(b) 2 clusters


(c) 3 clusters

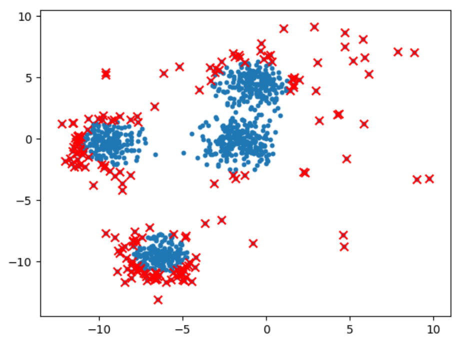
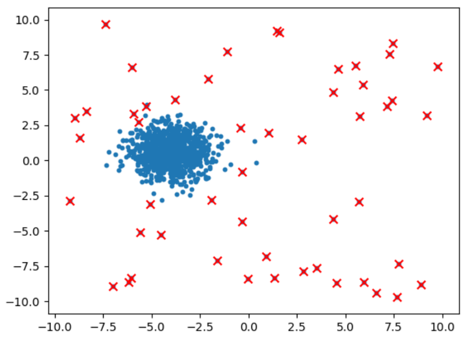
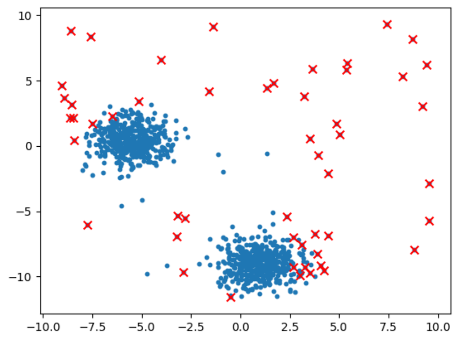
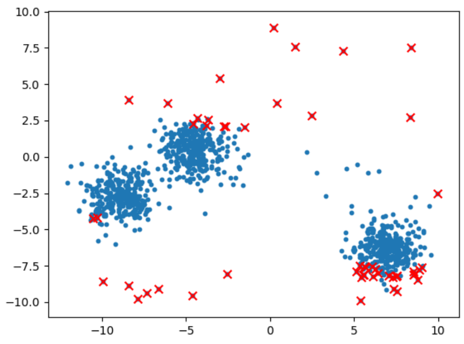

(d) 4 clusters

Figure 4.3: Isolation Forest performance vs. Clusterization


(a) 1 cluster


(b) 2 clusters


(c) 3 clusters

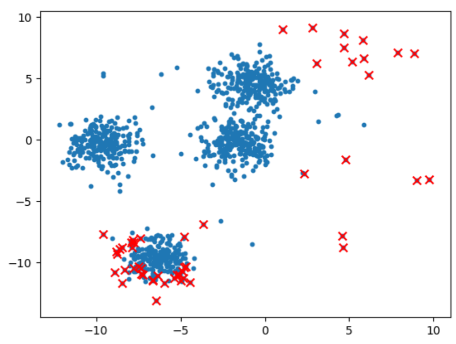
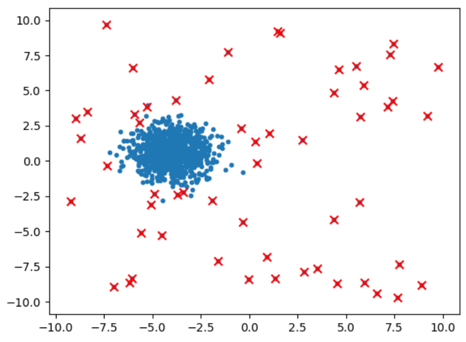
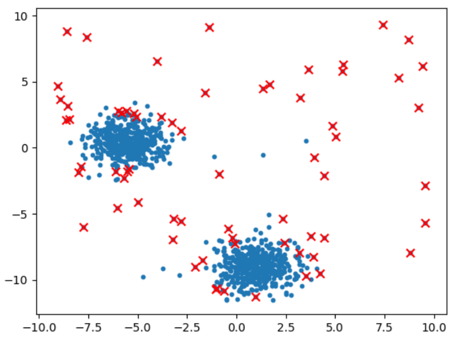

(d) 4 clusters

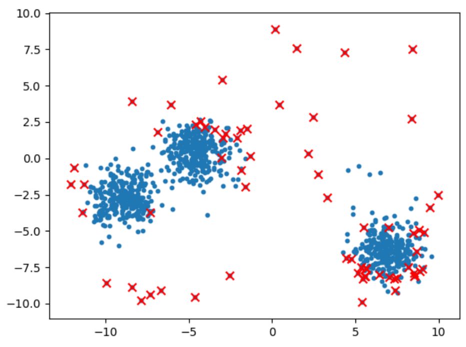
Figure 4.4: Half-Space Trees performance vs. Clusterization

Concerning the LODA algorithm, by observing the figure 4.5 one can infer that LODA is quite robust, but requires, like the HSTress, some initial data streamed in order to bootstrap its model (the more the better) and begin giving correct predictions.

Also, its hyperparameters need fine tuning, especially the threshold used and the `BucketWidth`. The author of the original paper [22] claims that the hyperparameters `BucketWidth` and `NumVectors` of LODA can be determined automatically, however this is true only for when sample data of reasonable size are available a-priori.


(a) 1 cluster


(b) 2 clusters


(c) 3 clusters

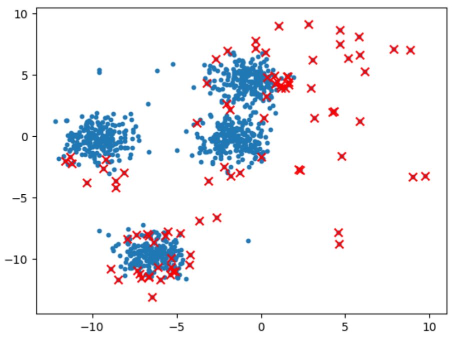

(d) 4 clusters

Figure 4.5: LODA performance vs. Clusterization

The results of each algorithm's performance is evaluated via measuring the F1 scores and Area Under ROC curve scores (AUROC), and are summarized in the graph 4.6.

In the graph 4.6 we can see that all algorithms exhibit a "drop" in the their accuracy scores, when the number of clusters in the dataset increases. Only Isolation Forest and LODA manage to keep a somewhat high AUROC score. The F1 scores are quite low in the case of hstrees and loda because of the bootstrapping time they need, which can cause them to miss some anomalies. The Mult. Gaussian algorithm

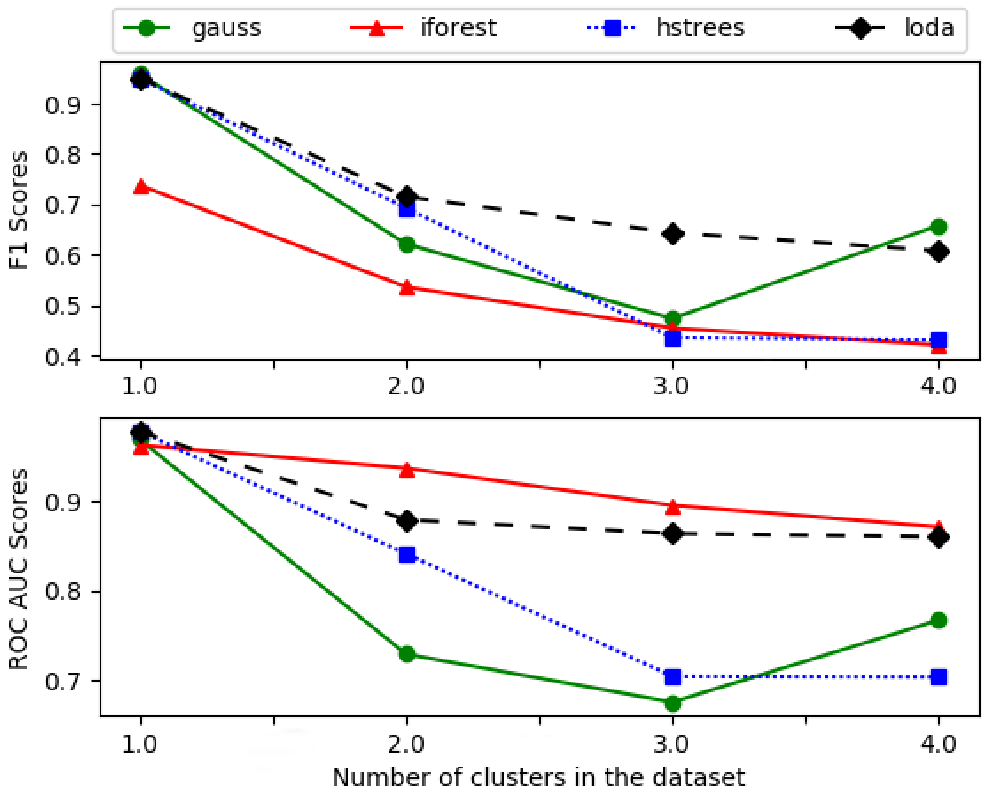


Figure 4.6: Detection metrics of different algorithms vs. Clusterization

also faces a drop because as we mentioned before, misses some positives (therefore since FP↑ $\implies$ F1↓, together with the Isolation Forest, that ends up flagging too many inliers as anomalies (FN↑ $\implies$ F1↓).

The next characteristic we are going to experiment on is the accuracy as a function of the number of features in the dataset. The synthetic datasets used are comprised of 1000 examples, 5% of which are anomalies, have clusterization number of 1, and the number of features ranges from 5 to 100 features.

The results of this experiment are shown in the graph in figure 4.7. The settings of the algorithms are the same as before (table 4.1), changing only the `NumFeats` parameter of LODA (which is necessary, since the model can't be built if `NumFeats` does not equal the actual number of features in the dataset).

As we can see, IForest and HSTrees have no issue with datasets of high dimensionality (they perform even better). LODA however seems to "struggle" with high dimensional data when it comes to prediction accuracy, but further investigation is needed in order to conclude, because it may be because of improper hyperparameter settings (especially `BucketWidth`). Finally, it is quite evident in the graph that the Mult. Gaussian algo is inadequate for high dimensional data. The F1 scores drop dramatically at approximately 20 features in our synthetic dataset. This happens because the probability calculated (which is a product of m factors, m being the number of features), becomes an extreme number (either too small, or too big, depending on the data), causing the process of finding a good threshold to be a difficult task. If for example we have a normalized dataset, in the range $[0, 1]$, and $m = 20$, the probability values can easily reach 10^{-20}.

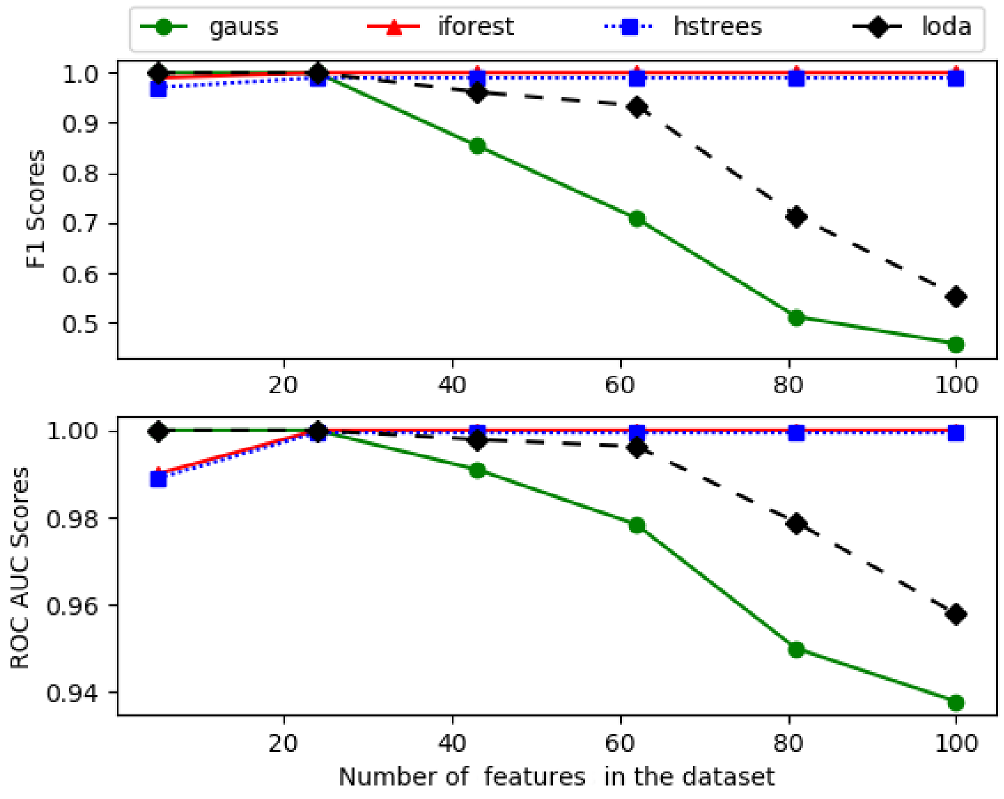


Figure 4.7: Detection metrics for different algorithms vs. Number of Features

4.3.3 Overall Time Efficiency

In this section we will compare experimentally the time complexity of each algorithm, both for the training phase and the inference phase.

The first experiment will be the measurement of the total time elapsed, in relation to the size of the dataset. For this comparison we used synthetic datasets, with one cluster, five features and the number of examples ranging from 1000 to 100000. The results are shown in figure 4.8.

For the Mult. Gaussian, the measured time is the sum of the seconds needed for training, inference, and evaluation (in order to find a good threshold value). For the IForest, it is the sum of seconds elapsed again for training and inference (the threshold value is constant, no need for calculation), and for the HSTrees it is the time for inference (since the "training" is done concurrently), plus the time to find a good threshold based on the scores of the predictions. Finally, for LODA the times shown are the times for inference (and also training which, like HSTrees, is done at the same time on-line).

In figure 4.8, it is obvious that all the algorithms have linear time complexity, and this characteristic is what makes them very scalable. Especially LODA and HSTrees require only one pass on the data for both updating their model and giving predictions, when deployed for on-line evaluation.

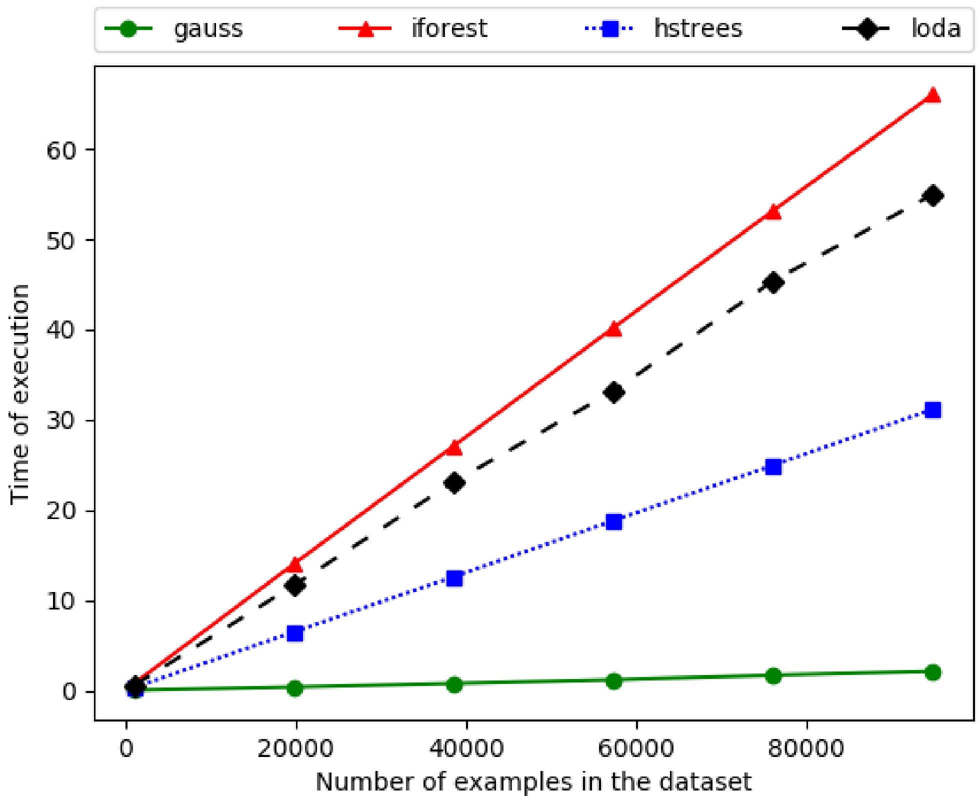


Figure 4.8: Total execution times for different algorithms vs. Size of Dataset

Next, we will examine how the number of features in the dataset affects the execution time. In figure 4.9 we can see that no algorithm is affected by the number of features, as the time needed stays constant when the number of features is growing.

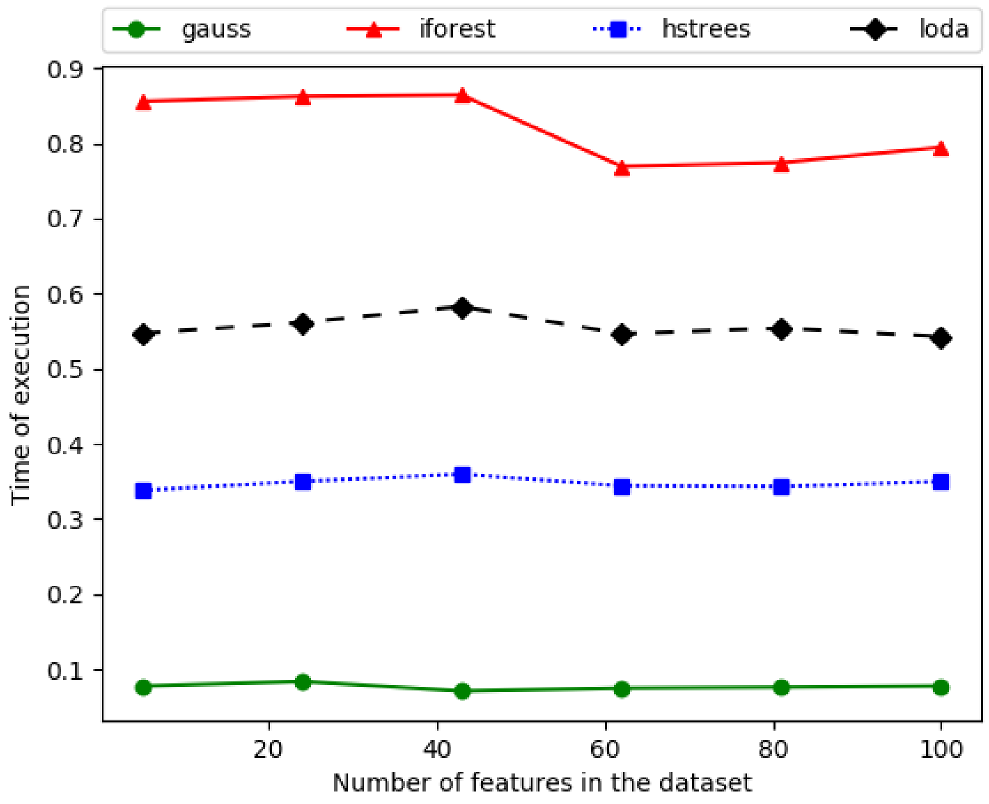


Figure 4.9: Total execution times for different algorithms vs. Number of Features

4.3.4 Evaluation on KDD Cup 99 Dataset

The synthetic datasets are good for understanding the behaviour of each algorithm, but when it comes to choosing an algorithm for a specific task or real world application, it is very important that we put our algorithms to the test and, using the correct metrics, decide which one performs best.

In our case, we have narrowed down our initial choices of Anomaly Detection algorithms by selecting those that are suitable for large scale on-line inference, and now we will investigate how they perform against the KDD Cup 99, a real dataset. As we mentioned in section 4.2, for the accuracy and efficiency tests we will use the HTTP and SMTP parts of this dataset.

For the HTTP dataset, we used the settings presented in table 4.2, and the measurements from the experiments are shown in table 4.3.

We can see that the Multivariate Gaussian algorithm achieved very high F1 and AUROC scores (0.97 and 0.99 respectively) and is also the fastest. This big difference in time of execution is because of many reasons:

- The Mult. Gaussian algorithm has a small model, as it doesn't use big ensem-

Algorithm	Settings	Algorithm	Settings
Gauss	Iter: 1000	IForest	Auto: True
			NumTrees: 100
			SubsampleSize: 256
HSTrees	NumTrees: 100	LODA	NumFeats: 3
	SizeLimit: 500		BucketWidth: 1
	MaxDepth: 10		NumVectors: 200
	WindowSize: 5000		TrainSize: 30000
	EstNumAnomalies: 3000		ProbSumBorder: -8.65

Table 4.2: Parameter settings for the HTTP KDD Experiments

Algo	F1 Score	AUROC Score	Total Time (s)
Gauss	0.9707	0.9900	2.566
IForest	0.5709	0.9716	46.425
HSTrees	0.8344	0.9907	86.458
LODA	0.1333	0.7667	35.856

Table 4.3: F1, AUROC Scores and execution times for each algorithm, on the HTTP KDD dataset

bles like those of the other algorithms,

- It is not recursive, like the IForest or HSTrees, and therefore it does not have the negative impact of the overheads that slow repetitive function calls introduce,
- Was entirely implemented with NumPy, which is optimized for matrix-based numerical computations at a low level, while the others, despite leveraging NumPy whenever possible, they suffer more from the high level and therefore slower Python runtime.

The results of HSTrees are quite satisfactory as well. LODA on the other hand performed very poorly, but it may be because of incorrect hyperparameter settings. If this is the case, the difficulty of setting the hyperparameters of LODA correctly in before, especially without having sample data (which is common is some applications), should be added to the negative characteristics of the algorithm.

Next we run the same experiments on the SMTP dataset. The algorithm parameters were set to the same values with those shown in table 4.2, with the only differences being the parameter `EstNumAnomalies` of HSTrees which was set to 170, and the parameters `NumFeats`, and thresholding values `TrainSize` and `ProbSumBorder` of LODA, which were set to the values 38, 1000 and -4.79 respectively. respectively. The results are visible in the table 4.4.

Algo	F1 Score	AUROC Score	Total Time (s)
Gauss	0.6833	0.9861	1.217
IForest	0.9626	0.9640	7.623
HSTrees	0.6891	0.9044	15.159
LODA	0.7123	0.8814	6.073

Table 4.4: F1, AUROC Scores and execution times for each algorithm, on the SMTP KDD dataset

In this dataset the best results are achieved with the Isolation Forest algorithm, which scores 0.96 for the F1 metric and 0.96 for the AUROC metric. Notable are also the results of Multivariate Gaussian.

4.3.5 Execution with Apache Flink

For our last experiment, we will measure the rate at which data can flow through our distributed Anomaly Detection system, also known as *throughput*. We executed each of the four algorithms on the cluster, using implementations in Scala, for various amounts of dataset size and cluster parallelism.

For the streaming process, a node reads the dataset stored as a file in its filesystem, and forwards each line of the file representing a data point, to the other parallel instances (which may be hosted in the same machine or across the network on another machine), the *mappers*. The mappers will then *map* every point they receive to their model (and also update it in the case of HSTrees and LODA), and then make a prediction based on this model anomaly or not). After that, they write their predictions on their local filesystems. The actual topology is the one shown in figure 4.10.

Because for our experiments we used a file as *data source*, the action of reading the file from the disk is done with parallelism 1. This fact, along with the fact that network I/O is faster than disk I/O, especially for this particular cluster that uses HDDs instead of SSDs for storage, leads to the creation of a *bottleneck* in our setup, restricting the max throughput.

To find out what is the max throughput value, we measured how long it takes for the whole dataset file (4898431 lines) to be read by a program. This action was timed at 23 seconds, therefore the max achievable throughput is about 212975 lines (or datapoints, since each line represents one example), per second.

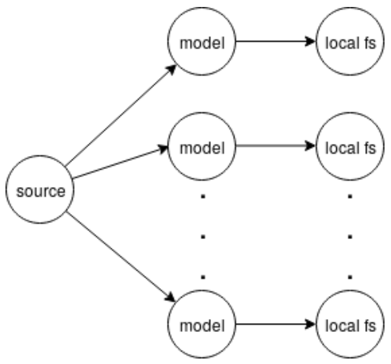


Figure 4.10: Topology used for on-line inference in Flink

The execution of our experiments in Flink was *pipelined*, meaning that the datapoints were streamed by the time they were read from the disk. Thus, the times measured to calculate the throughput were actual computation times, and the disk I/O operations did not affect the result in a negative way.

Concerning the first experiment, with variable dataset size, we kept the parallelism at the max possible value that could be achieved in a cluster of four nodes, with 2 vCPUs each, i.e. 8, and measured for each algorithm how much time it takes to process a stream created from the KDD dataset, using each time a different percentage of the dataset size (25%, 50%, 75% and 100%). Then we divided the number of tuples included in this dataset portions by the time the execution lasted to get the throughput, and we plotted the findings, which are shown in figure 4.11. The parameters of HSTrees were `NumTrees:50`, `MaxDepth:10` and `WindowSize:1000`, and LODA's parameters were set to `BucketWidth:10` and `NumVectors:100`. IForest used an ensemble of `NumTrees:100`, created with `SubsampleSize:256`.

We observe that the best throughput is accomplished with the Multivariate Gaussian, which manages to reach the maximum possible throughput we calculated earlier, about 212975 rows per second. LODA reaches 100000 rows second, HSTrees 70000 and IForest 96000. Again, the success of Multivariate Gaussian is because of the extremely small model size, and because it does not have to make any initializations. The other algorithms need more time to update their models, and also need some time to initialize it in the first place, especially IForest which has to load its model from an external file, as we described in Chapter 3 and build it.

By looking at figure 4.11, we also notice that the throughput is increasing along with the size of the dataset, while one might expect it to be constant. This increase is because the parallelism for this experiment was kept constant and at the maximum possible amount. As more nodes are added to the cluster, the total initialization time increases, and if the dataset is not big enough, a big amount of parallelism may add so much overhead that may not be worth it. In the realistic case where the data

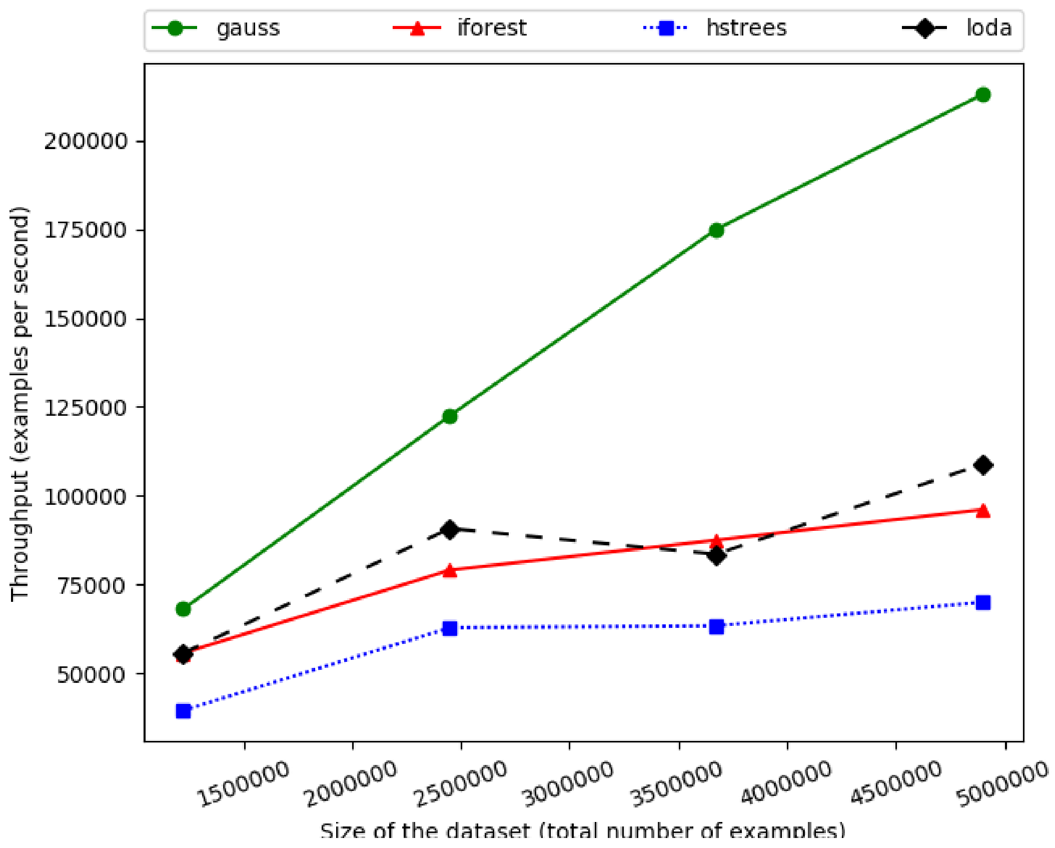


Figure 4.11: Throughput of different algorithms on Apache Flink vs. Size of KDD Subset

streams are infinite this is not a problem though, as the data never ends, but it is good to keep it under consideration when dealing with batch data at scale. In this experiment, where the stream was finite, if the dataset is small, the total overheads (mostly caused by the network I/O) increased the total time of execution, making the calculated throughput value drop. That is why we chose the last throughput measurement as the closest to the true throughput for each algorithm, because it corresponded to the largest dataset streamed through the system.

Next we will measure how parallelism affects the throughput. The settings are the same as before. The only change in the execution is the amount of parallelism in the cluster, that is essentially the amount of parallel *mappers* to use. The results are demonstrated in figure 4.12.

It is evident, and also expected, that by increasing the parallelism, the throughput of all algorithms also increases.

In conclusion, we can also perform a rough estimate of the latency of our system, i.e. the seconds needed for an example to go through and “come out” as a prediction from the models. For the Mult. Gaussian, this time would be about 4.7μs, for the

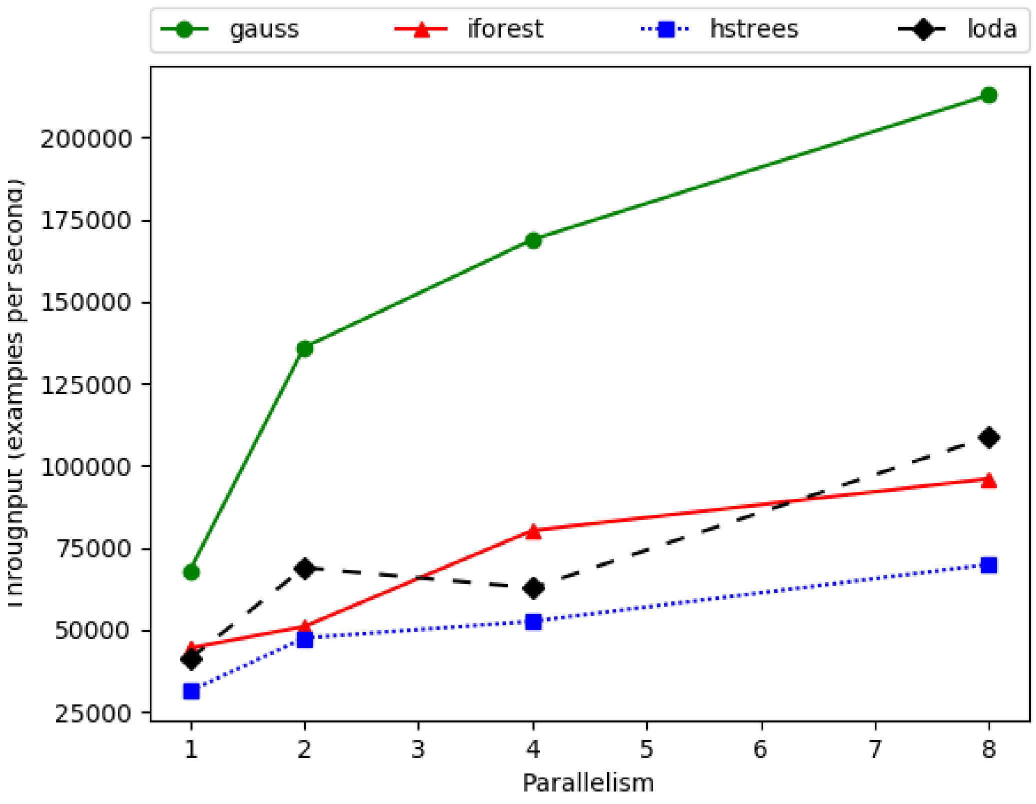


Figure 4.12: Throughput of different algorithms on Apache Flink vs. Parallelism

IForest 10.4μs, for HSTrees about 14.5μs and for LODA 9.2μs.

Chapter 5

Conclusions

5.1 Discussion

5.1.1 Comparison with Other Implementations

By the time of writing this thesis, no open-source implementations for Half-Space Trees and LODA exist, according to our best knowledge.

Concerning the Isolation Forest, a good open-source implementation is present in the scikit-learn Machine Learning library for Python. In terms of performance, when compared to the scikit-learn implementation, ours is slower, but yields the same ROC AUC scores. However, scikit's IForest does not offer a way to export the model in some form, while our IForest can export the trained model as a JSON that can be loaded by a Flink's (or other framework's) parallel instance to perform inference on streams of data. In addition, no implementation of IForest could be found in Scala.

For the Multivariate Gaussian algorithm, since it is a very versatile algorithm, implementations can be found in various forms, but no implementation exists using the DataSet or DataStream APIs of Flink. Also there is no implementation of the pure on-line version.

Flink does offer out-of-the-box some Machine Learning algorithms, in the Flink-ML library, but only one is for outlier detection, the SOS algorithm (Stochastic Outlier Detection), which is only implemented for batch processing and has a super-linear complexity of at least $O(n^2)$.

The code written for this thesis will be open-sourced in the near future.

5.1.2 Possible Extensions

Anomaly Detection System Improvements

When writing programs to run on a framework, one needs to be aware of the specific framework's features and the way these can be leveraged in order to achieve the best possible result. In the case of Flink, its official documentation [2] mentions a few ways that can make the execution run faster. The first way is by using function annotations that give Flink important high level information such as which fields of the tuples representing the data are forwarded unchanged during transformations, or not forwarded at all. Other optimizations include the use of tuple indexes instead of field names, reusing Flink object to relieve pressure on the garbage collector and more [3].

Also, in all our Flink experiments we used a single source and sink for our streams. In a realistic scenario, the streams of both input and output data are numerous. In stream processing applications, raw data are often collected in a message queue, then they are fed in a parallel manner into the stream processor, and after processing they are rewritten (also in parallel) in the message queue for persistent storage (historical data) and/or consumption by the actual application.

Thus, a nice addition that would make our setup a realistic system would be a message queue. A widely distributed message queue is Apache Kafka.

Another addition that would complete our system would be a distributed key-value store such as Apache Cassandra or Redis. Depending on the application, a join with historical data may be required. For example, in the case of Credit Card Fraud Detection, some necessary features the system would make use of would be the distance from the last executed transaction, or the time elapsed since then. This information must be acquired for each individual transaction, and the best way this can be done in a fast and scalable manner is through a low latency distributed key-value store.

Finally, a useful addition for large scale assessment would be the calculation of evaluation metrics online, directly in Flink. For example, the F1 Score can be calculated with the following transformations:

```
// Scala
labelledDataSet
  .map {
    row =>
      val flag = if (applyModel(row._1) < epsilon) 1 else 0
      val actual = row._2
      // (truePositives, falsePositives, falseNegatives)
      (flag & actual, flag & (~ actual), (~ flag) & actual)
  }
  .reduce { (x1, x2) => (x1._1 + x2._1, x1._2 + x2._2, x1._3 + x2._3) }
  .map { x => f1Score(x._1, x._2, x._3) }
  .collect().head
```

, where `epsilon` is the threshold, `row._1` is the example vector and `row._2` is the label with the actual value.

Algorithm Improvements

A good practice before running any of the four algorithms would be to do a PCA first (Principal Component Analysis), in order to find relevant features. This can lead to a decrease in the number of featues, making the algorithms run faster and be more precise in identifying outliers, especially the Mult. Gaussian algorithm, which was built on the hypothesis that features are independent. While this is almost never true in practice, a PCA run first would help us find all correlated and only use those that are actually unrelated.

For the HSTrees and LODA algorithms, there are some things that can be studied. For example a non-windowed version of HSTrees, or a windowed version of LODA following the paradigm of HSTrees might be interesting. One other addition to these two would be a *Contamination* parameter, like the one of IForest. If the percentage of anomalies in the dataset is known or at least possible to be estimated beforehand, then maybe a mechanism for adjusting automatically the threshold for both algorithms may be feasible.

5.2 Concluding Remarks

In this work, we explored Anomaly Detection, in the context of Big Data. Our requirements were on-line inference of anomaly scores, so that we could achieve detection in real-time, and also distributed, so that we could identify outliers at a big scale.

We chose four algorithms, namely Multivariate Gaussian, Isolation Forest, Half Space Trees and LODA, each having its own strengths and weaknesses:

- Multivariate Gaussian: Best scores in most synthetic and real datasets, super lightweight model, very fast, reached the maximum possible throughput when run on-cluster, easy to estimate a good threshold. Not very good for data in more than one clusters, high dimensional data, or data with non-independent features. (PCA recommended)

- Isolation Forest: Very robust, very good scores on real datasets, has “auto” mode -no need for estimating threshold-, strong against *swamping* and *masking* and has no problem with high-dimensional data. However, it can’t be trained in a distributed manner efficiently, and also, showed a high number of false negatives on multi-clustered data.
- Half-Space Trees: Purely on-line algorithm (model is updated on-the-fly), operates in windows. Good performance on real datasets. Suitable for evolving streams, has no problem with high-dimensional data. Hard to find a good threshold value, needs sample data (or at least a good estimation of the ranges of the values of each feature) or normalized data. Exhibited a bit high latency on-cluster when compared to the other algorithms, because it has a big model.
- Lightweight On-line Detector of Anomalies: Also purely on-line, but not windowed and runs slightly faster on-cluster than HS-Trees. Suitable for streams with missing values and for streams with *concept drift*. Nevertheless, without sample data it is very hard to find a good threshold and parameters. Not very good scores on the real datasets. Also, requires features to have approximately the same scale.

We also explored a promising engine for distributed stream processing, Apache Flink, and also Ansible, a tool that helps with the infrastructure when dealing with large clusters. Lastly, we suggested some possible extensions to our system and some improvements to the algorithmic setups.

Bibliography

[1] Ansible Documentation. `https://docs.ansible.com/ansible/latest/index.html`.

[2] Apache Flink Documentation. `https://ci.apache.org/projects/flink/flink-docs-stable/`.

[3] Flink Optimizations. `https://brewing.codes/2017/10/17/flink-optimize`.

[4] C. C. Aggarwal. *Outlier Analysis*. Springer New York, 2013.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2011.

[6] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[8] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[9] J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[10] C. B. Do. The Multivariate Gaussian Distribution. `http://cs229.stanford.edu/section/gaussians.pdf`, 2008.

[11] C. Ferri, J. Hernández-Orallo, and R. Modroiu. An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, 30(1):27–38, jan 2009.

[12] A. K. Ghosh, C. Michael, and M. Schatz. A real-time intrusion detection system based on learning program behavior. In *International Workshop on Recent Advances in Intrusion Detection*, pages 93–109. Springer, 2000.

[13] S. Hawkins, H. He, G. Williams, and R. Baxter. Outlier detection using replicator neural networks. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 170–180. Springer, 2002.

[14] D. J. Hill, B. S. Minsker, and E. Amir. Real-time bayesian anomaly detection for environmental sensor data. Citeseer.

[15] J. Janssens, F. Huszár, E. Postma, and H. van den Herik. Stochastic outlier selection. 2012.

[16] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream processing engines. *arXiv preprint arXiv:1802.08496*, 2018.

[17] G. Kayacık, A. Zincir-Heywood, and M. I. Heywood. Selecting features for intrusion detection: A feature relevance analysis on kdd 99. 01 2005.

[18] E. M. Knox and R. T. Ng. Algorithms for mining distancebased outliers in large datasets. Citeseer.

[19] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, dec 2008.

[20] R. Murphy. *On Tests for Outlying Observations*. Princeton University, 1951.

[21] A. Ng. Machine Learning on Coursera. `https://www.coursera.org/learn/machine-learning/resources/szFCa`.

[22] T. Pevnỳ. Loda: Lightweight on-line detector of anomalies. *Machine Learning*, 102(2):275–304, 2016.

[23] I. Saarinen. Adaptive real-time anomaly detection for multi-dimensional streaming data, February 2017.

[24] M. Salehi and L. Rashidi. A survey on anomaly detection in evolving data.

[25] S. C. Tan, K. M. Ting, and T. F. Liu. Fast anomaly detection for streaming data. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[26] K. M. Ting, G.-T. Zhou, F. T. Liu, and S. C. Tan. Mass estimation. *Machine learning*, 90(1):127–160, 2013.

[27] Wikipedia. Receiver Operating Characteristic. `https://en.wikipedia.org/wiki/Receiver_operating_characteristic`.

[28] K. Yamanishi, J.-I. Takeuchi, G. Williams, and P. Milne. On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. *Data Mining and Knowledge Discovery*, 8(3):275–300, 2004.