UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Data Skepticism in Practice

Online Anomaly Detection over Big Data Streams

Master's Thesis

Laura Rettig

Supervisor

Prof. Dr. Philippe Cudré-Mauroux

eXascale Infolab, Department of Informatics, University of Fribourg

Co-Supervisors

Dr. Mourad Khayati

eXascale Infolab, Department of Informatics, University of Fribourg

Dr. Michał Piórkowski

Swisscom AG, Bern

October 2, 2015

Abstract

Data quality is a challenging problem in many application domains. For Big Data, and in particular for high-velocity data streams, data quality testing is fundamentally different from previous approaches devised for relational data. This thesis describes and empirically evaluates the design and implementation of a framework for data quality testing over real-world streams in a large-scale telecommunication network. Our approach satisfies two conditions: generality—by using general-purpose measures borrowed from information theory and statistics that are applicable to various domains and data types—and scalability—through efficient and effective anomaly detection pipelines that are executed in a distributed setting. We propose two measures for dynamically detecting anomalies: relative entropy for detecting changes in the users' activity over time and Pearson correlation for detecting anomalies affecting individual data streams.

Our implementation leverages state-of-the-art streaming infrastructures, namely Kafka queues and Spark Streaming, as well as large-scale batch processing components such as HDFS and Spark. By combining both real-time and batch processing, we are able to detect anomalies at different temporal scales. By leveraging the spatial information given by the telecommunications network, we are also able to detect anomalies at different spatial scales.

We empirically evaluate our system and discuss its merits and limitations by comparing it to existing methods, showing its high accuracy and efficiency. Furthermore, we show that our system scales gracefully with larger volumes of data as it is able to parallelize its operations across large numbers of nodes in a cluster.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Nowadays, omnipresent sensing devices capture nearly every aspect of the world we live in. These devices transmit their detailed observations as vast amounts of data, the so-called *Big Data*. Examples of data sources include social networks, click streams, smart devices, and sensor networks. Following the quote by Naisbitt [31]—"We are drowning in information but starved for knowledge"—data by themselves are of no use. With the constant transmission of data from sensing devices, one of the major challenges today lies in making use of the obtained data. Currently, large quantities of data are stored, whereas only a small proportion of these data are analyzed. In an attempt to cope with big data, data science has recently been rising as a new field.

In 2014, Gartner's Hype Cycle, which provides an overview of the technology life cycle, maturity, and stage of adoption, positioned big data in the so-called "Trough of Disillusionment" [2]. This signifies that, after its initial position as a hyped technology, big data is maturing as skeptics point out limitations and businesses become more realistic about the usefulness and applications of big data. In this context, it has been discovered that not all data are good data and it is a nontrivial task to understand, measure, and guarantee the quality of data.

Since its rise, big data has commonly been defined by three characteristics, namely, the *3Vs of big data*: *volume*, *velocity*, and *variety* [22]. These characteristics required a paradigm shift in how data are treated compared to previous models for Relational Database Management System (RDBMS). The high *volume* of the data demanded for novel storage systems, leading to the widespread introduction of distributed database systems for storage as well as using statistical sampling techniques in processing the data. *Velocity* of the data demands for real-time processing capabilities. When retrieving terabytes of data on a daily basis, it is typically important to analyze data frequently, potentially reducing the need for storage by reducing the "raw" data to aggregated analytics. Data are of a high *variety* in that they are often unstructured and sparse. Data from different sources are inconsistent, such that additional information, e.g., metadata, is needed for resolution. These characteristics make it impossible to manually monitor each data item for quality and to impose trivial correctness constraints.

In recent years, further characteristic *Vs* have been added to define big data. Among these are *value* and *veracity* [37]. The *value* of the data is determined by the specific use case. With the growth in business use cases for big data, the data and the ability to manage them have become increasingly valuable. Big data generates value by enabling new analytics, such as insights on customer behavior, or by enabling new products [11]. The practical work for this thesis has been conducted at Swisscom, in the broader field of smart cities, where telecommunication monitoring data are analyzed to understand and predict mobility at an aggregated level. Hence our data are of value when reliably fulfilling customers' needs in gaining mobility insights. *Veracity* is closely linked to data quality and depends greatly on the trustworthiness of sources (raw data) and analytical pipelines (results). By being able to trace data and making them widely understandable, different consumers of big data products develop trust.

Given these characteristics, big data cannot be collected, monitored, and exploited in the same way as traditional data sources, e.g., RDBMS. Traditionally, every data point is controlled by quality constraints and represents a true aspect of the real world. Big data is commonly used for aggregated analyses where individual outlying data items have less weight than in databases containing "small data", such that new data quality models are required.

1.1 Motivation

Data quality is a challenging problem in many domains such as medicine, environmental monitoring, traffic monitoring, or IT infrastructures. Assessing the quality of the data requires the deployment of a number of fundamental data services including anomaly detection. Anomaly detection is an important element in monitoring data quality [15]–[18] and is hence a fundamental feature of many data-intensive systems. Using anomaly detection for measuring data quality follows the assumption that the majority of data are of high quality and non-anomalous, such that anomalies are directly linked to data quality problems.

Assessing data quality in real-time on data streams is important for being able to react quickly whenever real-time services are provided based on the data. Data quality is directly linked to trustworthiness. Being able to reason about data quality and its consistency is an important factor when communicating to potential customers of analytical platforms built on top of the data. Customers cannot blindly trust the quality of data. Different factors go into the estimation of the trustworthiness of data sources: the content and the processing method [10]. There is a high degree of control over the correctness of the processing, as it lies in the hands of those developing the analytics, and, albeit vulnerable to human error, can be tracked for comprehension. The quality of the data coming from the various sources lies beyond our control and is prone to various types of error. We therefore need to discover data quality on the existing data. The data that are used in this thesis originate from telecommunication monitoring probes. These probes transmit event feeds of high-level anonymized information about low-level activity on interfaces between the domain-specific components of the telecommunication network.

This thesis presents a framework for quantifying data quality in the context of such big data. The process of quantifying data quality is called Data Quality Testing (DQT). DQT aims at being able to quantify the quality of the available data, i.e., their fitness for fulfilling the tasks required by the data consumer and delivering trustworthy results. DQT is therefore also useful for increasing the quality of data by helping to identify weaknesses in the data pipeline. Data consumers have an interest in high data quality for creating more accurate and reliable insights, which can assist in gaining an advantage over competitors. It is not the goal, and hardly feasible, to create error-free data. However, measuring the error to determine the reliability of the implications drawn from the data and being able to intervene in cases where fundamental quality issues are present are important steps toward meaningful insights from big data.

Our DQT system needs to meet the following properties:

1. *Generality:* The system needs to be adaptable to different types of data, e.g., multidimensional or categorical.

2. *Scalability:* Since we are dealing with big data streams with a high velocity, we are interested in a system that scales to a larger number of machines for parallel processing.

3. *Effectiveness:* We would like to be able to quantify the statistical soundness of the DQT measures.

1.2 Contributions

This work introduces a DQT system for big data, both *at rest* and *on the move*, in the form of anomaly detection. Specifically, the main contributions are:

- An implementation of two measures lending from information theory and statistics—relative entropy and Pearson correlation—over Apache Spark [43] and its streaming library for detecting anomalies over both high-velocity streams and/or large volumes of data at rest.

- An empirical evaluation of our system showing the effectiveness of the two measures for detecting anomalies. The combined approach using these two measures enables the detection of different categories of anomalies. In particular, the relative entropy measure is well-suited for detecting gradual changes in data, while the correlation measure is more appropriate for detecting abrupt changes over data streams.

- A comparison of the proposed measures to state-of-the-art methods, namely, a naïve approach through counting the number of events during a timespan, as well as clustering using the k-means algorithm. We show that our measures outperform these techniques regarding both efficiency and accuracy.

- An empirical evaluation demonstrating the graceful scalability of our system to both an increasing number of nodes and increasing quantities of data.

1.3 Outline

This thesis is structured as follows: In Chapter 2, we provide an overview of related work in the fields of data quality, data streams, and anomaly detection. Chapter 3 explains the fundamentals of the technologies, the network infrastructure and the emerging data, and the measures that are used as part of the anomaly detection system. This system and the implemented processing pipelines are then described in Chapter 4. In order to empirically evaluate the anomaly detection system, Chapter 5 describes a series of experiments and their results. Chapter 6 discusses and compares the results from the evaluation, summarizes the important observations, and points out limitations of this work. Chapter 7 concludes the work and points out future directions in anomaly detection over data streams for DQT.

Chapter

2

Related Work

This chapter provides an overview over previous approaches related to DQT by means of anomaly detection over data streams, in the areas of:

- data quality: definition and approaches to measure and control the quality of data and in particular, difficulties related to big data (Section 2.1);
- data streams: definition and specific data structures and algorithms (Section 2.2); and
- anomaly detection techniques for time series and data streams (Section 2.3).

2.1 Data Quality

This work uses the definition of data quality as given by Olson [32], stating that data is of high quality if it satisfies the requirements of its intended use. Hence, data quality depends on both the data itself and the use case. In a first step, requirements toward the data are to be defined in order to proceed with evaluating their quality. Cong et al. [3] name *consistency* and *accuracy* as central criteria for data quality. While there are more dimensions to data quality, as described in the following section, our main interest lies in obtaining consistent data. We make the assumption that the vast majority of data are accurate in the sense that inconsistencies imply a break in accuracy.

2.1.1 Data Quality at Swisscom

A concept for the common understanding of data quality within Swisscom was defined by the Big Data & Business Intelligence Architecture division, lending from Olson's data quality definition [32]. This concept is only applicable to a limited extent, since it was created with data at rest, e.g., RDBMS, in mind, but general properties and dimensions of data quality will be described in the following.

Data quality can be measured syntactically by comparing the incoming data to certain specifications, schemes, or business rules. Triggers for alerts for data quality breaches include missing values or values that are not within a certain predefined range. Testing the quality from a semantic perspective is either done by automatically comparing all data to a "golden source" or by periodical manual examination of samples of the data.

Within Swisscom's data quality concept for data at rest, data quality is described along the following dimensions:

- *Accuracy*: degree of correctness to which the data represent the real world. In more detail, accuracy is divided into four subdimensions:
  - correctness: semantically and syntactically;
  - completeness and uniqueness: every real-world item is represented in an unambiguous manner in the data;
  - precision: data match the requirements for the use case;
  - integrity: related data points should also be captured and linked.
- *Extent of the data*: availability of expected attributes or aspects of the data.
- *Relevance*: extent to which the data fulfill the required task.
- *Trustworthiness*: reliability and accuracy of the data in relation to the real world.
- *Comprehensibility*: consumers' ability to access the data and make use thereof.
- *Timeliness*: usefulness of data at the time of arrival in business processes.

This latter aspect, *timeliness*, motivates the implementation of DQT over real-time data streams. Issues arise, however, with regards to *accuracy* and *trustworthiness* in stream processing, as validation thereof becomes more difficult in a scenario where not all data are available at the same time; instead, the quality needs to estimated on a proportion of recent data.

2.1.2 Data Quality in *Big Data*

The main difference between DQT on big data, including data streams, and DQT on traditional databases lies in the more diverse origins and dimensions of quality issues and anomalies in streaming data.

Kandel et al. [18] present an extension to databases through a visual interface to semi-automate the detection of anomalies and correction of data quality issues. They rely on statistical metrics including mutual information to present the most likely anomalous data on the user interface. Due to the involvement of a user to verify the detected anomalies, this semi-automated approach is not scalable and therefore not applicable to our scenario of real-time anomaly detection on high-velocity streams.

Geisler et al. [13] propose semantic models, incorporating domain knowledge, as an ontology-based approach to measuring data quality on data streams in a traffic information system. They compute a set of features as data quality metrics for individual data points. As another model-driven approach, Mazon et al. [28] introduce a system to annotate linked open data with a set of data quality criteria for the purpose of data mining. Similarly to Kandel et al. [18], their work does not fully automate the process, but merely returns metrics to the user, who in turn has to make their own interpretation. Furthermore, these approaches apply only to data at rest.

As a possible way to resolve the issue of checking data quality given the variety in big data, Korn et al. [20] take a probabilistic approach to monitoring network traffic feeds and comparing input data against constraints. The authors propose a system that uses *probabilistic, approximate constraints* to continuously monitor data quality. Their constraints include, as parameters, tolerance levels and likelihood of violation of the constraint. The parameters are configured based on the statistical properties of an analyzed set of training data. Both the approach by Korn et al. [20] and by Kandel et al. [18] are implemented as extensions to large-scale databases providing data quality information and not on data streams.

Klein et al. [19] propose a data stream metamodel which provides data quality information throughout the entire pipeline from the sensors to the applications using the data. Similarly to the work by Kandel et al. [18], their construction processes for the models integrate a high degree of domain knowledge and are thus less general.

In many applications, sensor data falls into the area of big data (as defined in Chapter 1), transmitting at high velocity from many different sensor locations. These sensors are unstable and highly vulnerable to failures, leading to the production of incorrect data, which cannot be monitored manually due to their volume and velocity. Data quality problems in data feeds may have any number of reasons, including sensor failures, failures in the data pipeline, and abnormal phenomena at the point of origin of the data stream. Consequently, data quality problems manifest themselves in the data in a variety of ways and a simple rejection of items on a stream leads to a loss of potentially valuable information about problems that need to be addressed. Defining integrity constraints for each possible error source is not scalable [30] and global specifications of the semantics are unlikely to fit the entire data [36].

Using various sensors as data sources, Huang et al. [16] introduce a data quality process in the area of virtual metrology. Their data preprocessing consists of a completeness check, verifying the integrity and the accuracy of raw input data, normalization to a schema, and reduction to remove redundancies. Due to the domain knowledge, missing data can be inferred. These steps lead to the correction of faulty data points and an improvement of the data quality with a generated reliance index. Dereszynski and Dietterich [9] take an approach to data cleaning using a *Dynamic Bayesian Network*, by this automating a manual quality assurance process in providing ecological data to scientists from a multitude of hardly accessible remote sensors. They focus on creating a model that is highly accurate in detecting normal behavior. Hence anomalies from faulty sensors and not actual changes to the recorded data are detected as data that do not fit the normal models given the learned patterns. As a result of the accurate normal models and the known previous data in case of anomalies, Dynamic Bayesian Networks also provide the ability to infer the correct values for anomalous periods. As our intention is not to improve the data quality but only to provide a measure for it, we do not try to correct any data on the streams.

2.2 Data Streams

Streams are defined differently depending on the application. It is therefore important to define how streams are interpreted in the context at hand [26]. In our context, data streams are interpreted as unbounded sequences of tuples with n attributes, $[(a_1, a_2, \ldots, a_n), (a_1, a_2, \ldots, a_n), \ldots]$, generated continuously over time. In a strict sense, data streams imply that only one data point is observable at any time. In many models, including our system, however, it is assumed that a small number of recent data points can be temporarily stored to an aggregate window of recent data, such that for example sliding window aggregate functions can be computed over the most recently arrived tuples [1]. Nevertheless, processing times for streams are typically bounded [13].

Streams frequently have the property that data arrive at a high velocity, posing problems in the the areas of transmitting input to a program, applying functions to large input windows, and storing data, both temporarily and long-term [30]. Hence, a substantial amount of work has been put into data structures and algorithms with sublinear complexity and storage requirements. Such sublinear techniques commonly employ probabilistic methods with bounded errors. Probabilistic methods are common when handling big data; in a similar sense, Korn et al. [20] took a probabilistic approach to DQT, as outlined in the previous section.

Statistical metrics and probabilistic data structures that represent sliding windows in streams have been proposed for summarizing streams. Datar et al. [7] introduce approximate stream summary statistics for sliding windows. Since regularities in streams may evolve over time, the issue of data decay is handled by giving more weight to recent objects, aggregating previous windows, and eventually discarding older data. The authors store information using *exponential histograms.* This data structure uses timestamps as the bins and the count of an item in the stream as the value for each temporal range. While their work is suitable for computing approximate statistics with bounded errors to summarize aspects of the content of a stream, they do not address the issue of detecting change.

Frugal streaming was introduced by Ma et al. [25] providing first-order statistics over data streams. These *frugal streaming* algorithms are able to treat streams one item at a time, requiring no memory of previous data and only a maximum of two pieces of information are maintained in memory. Flajolet et al. [12] proposed the *HyperLogLog* structure, a sketch suitable for counting distinct elements with bounded errors in a single pass over the data, making the algorithm highly suitable for stream data. While very simple and efficient, both approaches are restricted to streams of a single dimension.

Papapetrou et al. [33] introduce the *ECM-sketch*, a technique that is suitable for summarizing streams and for answering complex queries over data streams. Their *ECM-sketch* combines the capabilities of stream summary through key-based counting in a large key space (similar to the count-min sketch [5]), but extends them with exponential histograms for synopses of sliding windows. Unlike our work, *ECM-sketch* cannot use deterministic data structures, e.g., hashmaps, due to the potentially very large number of distinct items (keys) for which a counter (values) has to be maintained.

2.3 Anomaly Detection on Time Series Data and Data Streams

In this thesis, we define and identify data quality issues as anomalies, i.e., deviations from the expected model of the data. Related work on anomaly detection for time series data can also be applied to data streams. While time series do not require real-time systems, both time series and data streams provide in fact temporal data, as data streams naturally carry the notion of time [33] (either by means of time of arrival of a data point or from a timestamp associated with it). A number of techniques have been proposed to detect anomalies in multidimensional data streams or for multidimensional time series data.

A general method for detecting anomalies in datasets consisting of distributions is proposed by Lee and Xiang [23]. The authors use relative entropy amongst other information-theoretic measures to detect anomalies. Their measures are suitable for describing the characteristics of a dataset, but they do not address the data stream notion, requiring real-time computability. Based on the proposed information-theoretic measures, Dasu et al. [6] present an approach to detect sudden changes in multidimensional data streams. In their approach, multidimensional stream instances are represented as *kdq-trees* (a combination of *kd-trees* and *quadtrees*), while relative entropy is used as a similarity measure. To detect changes on unknown distributions, the method resamples the data from one window using the so-called bootstrap technique in order to obtain expected distributions of the data. The relative entropy between the distributions gives a bound for the relative entropy between different windows (under the assumption that the data originate from the same distribution), allowing for a statistically sound detection of significant changes. The authors propose two different window comparison models. The first model compares adjacent windows, which is well-suited for detecting abrupt changes. The second model compares a sliding window to a previous window, which is convenient to detect more gradual changes. We use similar techniques to measure changes between successive time windows over multi-dimensional data streams, which will be introduced in Section 3.2.1. However, we do not rely on complex and multidimensional data structures that would be very difficult to distribute and efficiently update on clusters of machines.

Cormode and Muthukrishnan [4] define *deltoids* as items where the computed metrics indicate that significant change took place in monitoring network traffic data. These deltoids are probabilistic metrics designed to use little space, short update times, and to produce accurate results based on fixed thresholds. The authors distinguish between different variations in data streams—absolute, relative and variational—and maintain deltoids for each type. Their approach is limited to streams of a single dimension and requires a pre-configuration of the parameters that are used to detect deltoids leveraging training data. Our proposed solution, on the other hand, aims at being more general-purpose without any preprocessing.

Zhang et al. [46] propose a solution that detects outliers in multidimensional data. The proposed approach performs anomaly detection by measuring the distance of a data point in various subspaces. The authors show that for multi-dimensional data, changes may be observable on one dimension, over a subset of dimensions, or overall. However, the proposed techniques based on indexing and subspace pruning are not applicable to real-time scenarios due to the high number of iterations over the data.

Li and Han [24] also address the problem of detecting anomalies in subspaces of multidimensional data by introducing the *time-series data cube* as a new data structure capable of handling the multidimensional space. Using this data structure, they are able to identify the subspaces that are most likely to be anomalous. To this end, they measure the entropy for each attribute and consider attributes with low entropy, i.e., attributes having mostly homogeneous values, to form part of subspaces in which anomalies are easily detectable. By selecting likely anomalous subspaces, they elude the curse of dimensionality and avoid having to search for anomalies in every possible subspace. The authors apply their technique to detect anomalies in synthetic data with anomalous time series, defining four different kinds of anomalies: trend, magnitude, phase, and miscellaneous. However, their solution works only for larger fluctuations and is not suitable in case of more subtle differences, which is the case for most of the real-world applications.

Anomaly detection techniques have also been proposed for strictly temporal data. Gupta et al. [14] present an overview of anomaly detection on various kinds of temporal data. They define anomalies as outliers and present detection methods for both the discrete and the continuous cases. They distinguish anomalies between singular anomalous points in time and anomalous patterns over time. It is pointed out that, regarding multidimensional data, changes may be observable over any subset of dimensions, a topic that is also being addressed in other works [24], [46]. While we do not directly search for anomalies in subspaces, we do consider different dimensions of the data in summarizing the streams, allowing us to detect anomalies that occur in the respective dimensions. The overview by Gupta et al. presents a wide array of different techniques, but it also mentions that there is a great deal of different problems that can be addressed by detecting outliers on time series data and that solutions need to be adapted to meet the needs of specific problems. For stream data, the use of models that update and decay over time is suggested. We take this into account by comparing to recent data when computing measures over the stream, instead of comparing to a fixed set of historical data.

Young et al. [41] detect and classify emergency and non-emergency events using annotated telecommunications network data, specifically, call detail records. Similarly to our work, they compare normal and anomalous days to detect deviations from a baseline representing average behavior. The known events in their dataset are detectable when plotting the call volume throughout a day for the anomalous event compared to an average for this day of the week. They observed that events change the users' activity at the location of the event, such that the difference—in terms of activity profile—to nearby cells, where activity is as normal, increases. We will compare our approach against this technique in Section 5.1.3. Unlike our proposed system, they use a metric that observes the anomaly only at the closest cell tower to the known event. Their work uses autoregressive hidden Markov models in order to classify timeframes and detect the precise onset of an event. Furthermore, the applied matrix factorization is computed on data at rest and not in real-time, unlike our high-velocity streams.

Wu and Shao [40] apply an autoregressive process to detect sudden changes between adjacent windows of network traffic data. Their use of moving windows allows real-time anomaly detection. However, their model is limited to detecting major and sudden changes, such as in denial-of-service attacks to a network and is not useful for detecting finer variations.

Clustering algorithms are frequently used to detect outliers or anomalous instances which have been assigned to anomalous clusters. In their survey of anomaly detection techniques for temporal data, Gupta et al. [14] note that different clustering algorithms, such as k-means, can be used to detect point outliers, as well as to create dynamic models for anomaly detection in streaming. Münz et al. [29] detect anomalies from network monitoring data as part of an intrusion detection system by using the k-means clustering algorithm. Instances are created by computing features on the traffic data per time interval. k-means forms k distance-based clusters based on unlabeled training data and assigns normal and anomalous instances each to a different cluster. In their setting, k is configured to 2, in order to assign normal and anomalous instances each to a different cluster.

The clusters' centroids are then deployed in order to classify new instances as either normal or anomalous. This is a highly generic approach that is fit for many scenarios. We will be comparing our technique against anomaly detection via k-means clustering in Section 5.1.3.

Chapter 3

Background

For a general overview, this chapter first provides a detailed introduction to the various open source technologies that are being used in the implementation of our DQT system (Section 3.1). This is followed by a description of the two measures lending from information theory and statistics, namely, relative entropy and Pearson correlation, which are leveraged in the implementation of the anomaly detection system (Chapter 4), in Section 3.2. In order to provide details on the specific domain context at Swisscom, Section 3.3 gives an explanation of the telecommunication network monitoring data sources and formats.

3.1 Technologies

This thesis focuses on the real-time processing of data streams. Multiple open-source streaming platforms have emerged in recent years, including Apache Storm[1], Apache Samza[2], Apache Spark's Streaming library[3], and Apache Flink[4]. Depending on the platform, different abstractions of the stream are used. Either each incoming data point is treated individually (in which case storing data, for example to perform aggregation over short periods in time, has to be implemented in the stream processing application), or data streams are abstracted to micro-batches, which perform processing after every defined interval. Further differences lie in the parallelization of the processing, as these streaming platforms are commonly distributed across multiple machines. We distinguish between *task parallelism* and *data parallelism*. With *task parallelism*, as implemented in Storm, different tasks are executed in parallel on multiple machines over the same data; i.e., each machine receives the data and is responsible for carrying out a specific task. In contrast, *data parallelism* implies that the same tasks are applied for each piece of distributed data; i.e., each task is performed multiple times in parallel on

[1]http://storm.apache.org
[2]http://samza.apache.org
[3]http://spark.apache.org/streaming
[4]http://flink.apache.org

multiple machines storing partitions of the data. This project uses Apache Spark [43] and Spark Streaming [44], the latter offering real-time processing in the form of micro-batches with data parallelism.

3.1.1 Apache Spark

Apache Spark is a general-purpose engine for large-scale data processing. Spark is part of the Hadoop[5] ecosystem and, being purely an engine for computing, not a database, can be connected to an array of data storage systems, e.g., the Hadoop Distributed File System (HDFS). Spark offers several advantages over MapReduce [8], including faster in-memory execution, especially for cases where multiple passes are made over the same data (such as when multiple stages of transforming, mapping, and reducing are applied to the data). This is attributed to the fact that Spark maintains data in memory, whereas in MapReduce pipelines, intermediary results are written to disk. Spark further provides a higher-level Scala Application Programming Interface (API), greatly facilitating the expression of complex processing pipelines.

Spark can be deployed either standalone, or over a cluster manager such as the so-called Yet Another Resource Negotiator (YARN). Applications require a defined amount of memory and in certain cases disk space on a defined number of nodes. These parameters are configured when starting a Spark application. YARN manages resources in a cluster and is responsible for allocating nodes and isolated memory partitions *(containers)* to an application. Figure 3.1 displays the components of a Spark deployment using YARN. The Spark driver runs on a local client application and connects to the Spark application master in the YARN cluster. The application master requests the required resources to run the job and allocates the task to the executors.

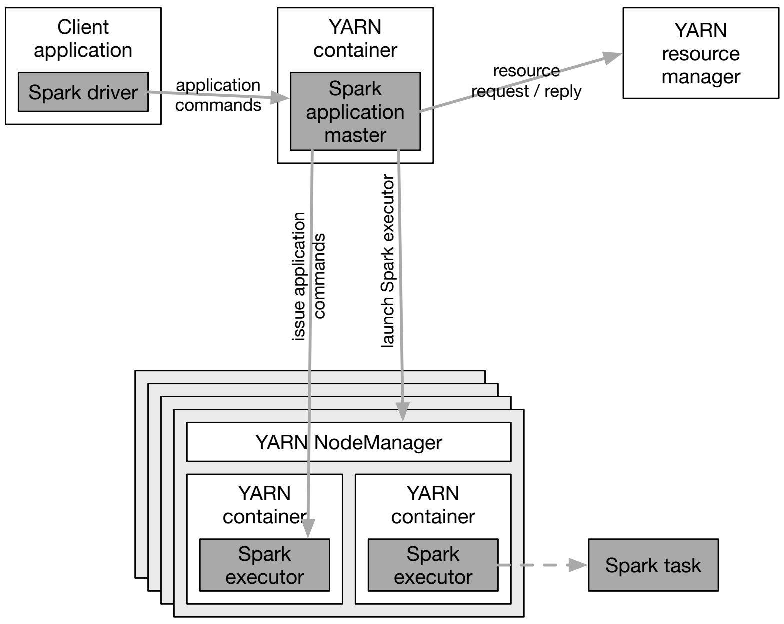


Figure 3.1: Components and infrastructure of Spark running on a YARN cluster [35].

[5]http://hadoop.apache.org

Spark's main abstraction are Resilient Distributed Datasets(RDDs) [42] for representing distributed datasets. An RDD is an immutable abstraction of distributed data; materialization of the data is done in a lazy manner. Each executor maintains a proportion of the data in memory. In batch processing mode, RDDs are created by loading data, for example from HDFS, or as a result of transforming another RDD. Deterministic transformations such as `map`, `filter`, `reduce`, and `join` can be applied to an RDD, yielding a new RDD. For each RDD, the sequence of operators is organized in a Directed Acyclic Graph (DAG) representing the data flow. This lineage of an RDD enables fault tolerance; lost partitions can be recomputed by re-applying the lost transformations on duplicated data following the nodes in the DAG. Since RDDs are evaluated lazily, the transformations are only applied when materialization becomes necessary. Data are materialized through actions such as `collect` operations for obtaining all data in one central location—the Spark driver—e.g., for printing to the console. These transformations are then applied in a manner that minimizes data shuffling between the executors.

In practice, Spark jobs are split into stages, reflecting the transformations in the DAG, where as many operations are executed in the same stage as possible. Any operator that can be applied locally on the data, i.e., individually for each physical partition, is executed locally in the memory of the executor for the data which it maintains. Examples of such local transformations include `map` and `filter`, which do not require any knowledge of other data partitions, or any shuffling of data between executors. Whenever data are shuffled and moved between partitions, such as for `reduce`, `sort` or `join` operations, a new stage is created. Stages are transparent and can be viewed for debugging purposes in the application by printing the RDD's `DebugString`. The `DebugString` essentially shows the RDD's associated DAG, giving instructions for its materialization.

As a minimal example of how functions are applied to data in Spark, Listing 3.1 demonstrates an implementation using the Scala API in the interactive Spark read—eval—print loop (REPL). This (simplified) pipeline loads data from HDFS into an RDD, prepares the data by mapping and filtering, then reduces and aggregates the key-value-RDD. This pipeline yields the distribution of the event counts per timestamp for the loaded data, making use of three attributes coming from each event tuple (cf. Section 3.3.1 for an explanation of the form of the data): the timestamp, the anonymized user ID (International Mobile Subscriber Identity (IMSI)), and the event ID (call type). The transformations are applied in three stages, as can be seen in Listing 3.2, which shows the `DebugString` for the RDD `eventsATypes`: Each `ShuffledRDD` creates a new stage for the data being shuffled between executors (caused by `reduceByKey` and `groupByKey`), while `map` and `filter` are applied locally for each partition in the `MapPartitionsRDD`. Although the RDD `eventsATypes` is newly created, it also includes the instructions for its parent RDD `eventsA`, since there is no materialization when creating a new RDD from an existing one.

Spark Streaming

Spark includes a streaming library called Spark streaming. Spark Streaming provides the ability to consume real-time data from various sources, including Apache Kafka (cf. Section 3.1.2). Stream processing is based on micro-batch computations and introduces a second core abstraction, Discretized Streams(DStreams) [45]. DStreams are continuous sequences of RDDs, with one RDD containing all the data belonging to one micro-batch of a fixed duration. Many of the

```scala
val eventsA: RDD[(DateTime, String, Integer)] = IOUtils.readAvroRawFile("[...]/daily/2015/
    06/30/*", classTag[A])(sc).
  map(event => (new DateTime(event.start_time_seconds*1000L),
    event.imsi.asInstanceOf[ByteBuffer],
    event.getCallType)).
  filter { case (t, imsi, e) => imsi != null }.
  map { case (t, imsi, e) => (t, new String(imsi.array()), e) }

val eventsATypes: RDD[(DateTime, List[Int])] = eventsA.map { case (t, imsi, e) => ((t, e),
    1) }.
  reduceByKey(_ + _).
  map { case ((t, e), count) => (t, (e, count)) }.
  groupByKey().
  map { case (t, counts) => (t, counts.toList.sortBy(_._1).map(_._2)) }
```

Listing 3.1: Pipeline for creating event type histograms, i.e., obtaining the count for each event type within a given time.

```
(20) MapPartitionsRDD[42] at map at <console>:74 []
 | ShuffledRDD[41] at groupByKey at <console>:72 []
 +-(20) MapPartitionsRDD[40] at map at <console>:71 []
    | ShuffledRDD[39] at reduceByKey at <console>:70 []
    +-(20) MapPartitionsRDD[38] at map at <console>:69 []
       | MapPartitionsRDD[5] at map at <console>:74 []
       | MapPartitionsRDD[4] at filter at <console>:73 []
       | MapPartitionsRDD[3] at map at <console>:70 []
       | MapPartitionsRDD[1] at map at IOUtils.scala:75 []
       | [...]/daily/2015/06/30/* NewHadoopRDD[0] at newAPIHadoopFile at IOUtils.scala:74
           []
```

Listing 3.2: `DebugString` demonstrating the steps and the three stages of execution for creating event type histograms.

functions available for RDDs are also available for DStreams, abstracting away the individual processing of RDDs during streaming, such that transformations can be directly applied on DStreams. In this case, any transformation applied to the DStream is, in fact, applied to each micro-batch's RDD individually by the Spark engine when the data for this micro-batch arrive.

Figure 3.2 shows an incoming DStream of events as a sequence of micro-batch RDDs and the application of operators to each RDD. Since both RDDs and DStreams are immutable, the output of applying a transformation to a DStream is a new DStream, representing a continuous sequence of transformed RDDs. The underlying execution engine, the Spark engine, is the same for both streaming and batch modes. The execution engine obtains one RDD from the DStream per micro-batch time interval and applies the transformations directly to the RDD. DStream operations are categorized into transformations and DStream-specific output operations. Examples of transformations include `map`, `reduce`, and `window`. Windowing groups together multiple micro-batches into batches over longer periods of time, also shown in Figure 3.2, where non-overlapping windows are created at multiples of 2 of the micro-batch duration and written to a new DStream. Output operations are performed on each RDD in the stream and include printing results or saving to disk, leading to transformation of the data along the DAG and materialization of the current dataset in the stream. Since streams are also materialized lazily, an output operation is required to start the streaming. Otherwise, no computation takes place.

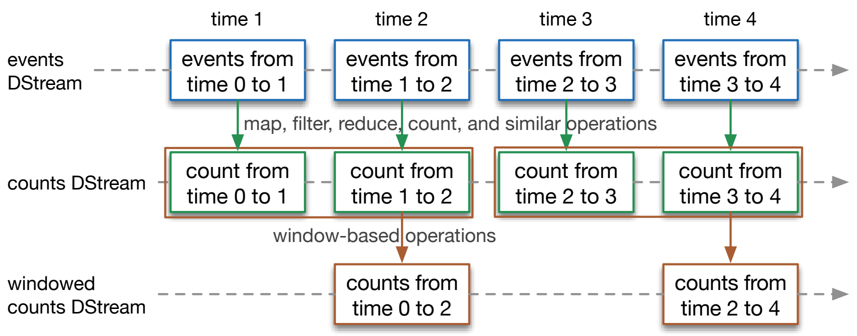


Figure 3.2: DStream micro-batch model. Each box corresponds to one RDD. Operators that are applied directly to each RDD in the DStream and window operations that group together data from multiple RDDs over a period of time transform one DStream to another (adapted from the Spark Streaming Programming Guide [38]).

A DStream's RDDs are processed sequentially in the order in which they arrive. It is important that any processing terminates in less than the micro-batch interval duration. Figure 3.3 shows the consequences of different processing times. If the processing takes less than the batch duration, one micro-batch can be processed while the receivers collect the data for the next micro-batch. Once the next micro-batch is ready to be processed, the previous processing has completed and the computational resources are available. If the processing takes longer than the batch duration, the previous micro-batch is still being processed while the next one becomes available, meaning that these data have to be stored intermediary while at the same time receiving again another micro-batch. This way, older data add up and increasingly delay the processing. Eventually, processing will no longer be possible, because old stream data, which have not yet been processed, had to be removed in order to receive and store newer data.

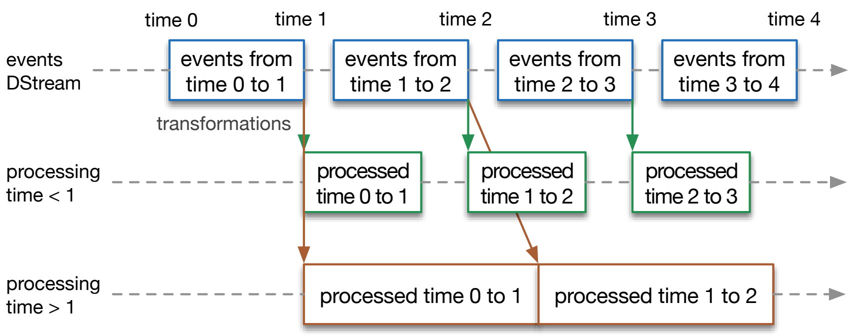


Figure 3.3: Timeline of data receiving and processing and the impact of different computation times. Processing starts once all data are received for the previous micro-batch. Since RDDs are processed sequentially in the order of arrival, if the processing takes longer than the duration of one micro-batch (here: duration 1), increasingly more RDDs need to be stored until they can be processed.

Code Reuse Between Spark and Spark Streaming

As part of the code can be reused with minimal adaptation between batch and stream processing, Spark is well-suited for cases where both batch and streaming data are to be jointly processed, or where similar pipelines exist for both real-time and batch processing. Listings 3.3 and 3.4 show examples of Scala code produced in this project, once for batch and once for streaming, highlighting the similarity of the two programming models. On line 2, a `filter` operation is applied to the DStream in the same way as to the RDD. While in streaming, it is sufficient to count the distinct elements in the current RDD, in batch mode, the RDD contains the data for the entire day and therefore, it is mapped to key-value pairs with the timestamp as key and the anonymized user ID, IMSI, as the value. After further `map` and `filter` operations, the `transform` operation on line 6 in Listing 3.3 applies an RDD-specific function to each RDD in the stream and then performs a `count`, one of the provided high-level API functions, on each dataset in the stream. In batch, on the other hand, the API provides a function for directly computing distinct values per key (here: timestamp), which uses the probabilistic HyperLogLog data structure [12] with an error bound of 0.05. While windowing can be performed directly in streaming, it requires another mapping of the timestamp in batch and has been omitted for the sake of brevity in the example in Listing 3.4.

```
  val countStream: DStream[Long] = inputStream.
2   filter(_.get("current_cell_lac").asInstanceOf[Int]==2020).
    map(event => event.get("imsi").asInstanceOf[ByteBuffer]).
4   filter(imsi => imsi != null).
    map(imsi => new String(imsi.array()).toLong).
6   transform(rdd => rdd.distinct).
    count().
8   window(Seconds(windowDuration), Seconds(windowDuration))
```

Listing 3.3: Counting the number of distinct anonymized user IDs for each micro-batch in streaming, then windowing over multiple counts.

```
  val countsPerBatch: RDD[(DateTime, Long)] = inputEvents.
2     filter(_.getCurrentCellLac==2020).
      map(event => (new DateTime((event.start_time_seconds-(event.start_time_seconds%
          countBatch))*1000L),
4       event.imsi.asInstanceOf[ByteBuffer])).
      filter { case (t, imsi) => imsi != null }.
6     map { case (t, imsi) => (t, new String(imsi.array())) }.
      countApproxDistinctByKey(0.05)
```

Listing 3.4: Counting the number of distinct anonymized user IDs per period of `countBatch` seconds in batch.

3.1.2 Apache Kafka

Apache Kafka[6] is a high-throughput fault-tolerant distributed publish/subscribe system, enabling real-time access to data streams. Kafka is run as a cluster and handles multiple streams, or message feeds, called *topics*. *Producers* publish messages to a Kafka topic and *consumers*, such as our application, subscribe to topics and process the message feeds. Topics may be partitioned over multiple machines, called *brokers* in the Kafka cluster, which benefits the consumers in that they can receive data in parallel. Fault-tolerance is achieved through replication,

[6]`https://kafka.apache.org`

where the replication factor can be set individually for each topic. The Kafka clusters retains the data for either a certain period in time or until a maximum size is reached; in the current configuration, our retention period is set to 2 days.

The joint use of Spark Streaming and Kafka provides *at least once* and *exactly once* processing guarantees for received records. A consumer application identifies itself to the Kafka cluster by its *group ID*. Kafka stores the timestamp of the last consumed data for each consumer group. This way, if a continuously running real-time application fails for a short time (less than the retention period), it can retrieve the lost data by asking the Kafka cluster to send all data since the last consumption timestamp.

Kafka, as part of the so-called Firehose, which is described in Section 3.3.2, is used purely as a data provider in this project, being consumed through Spark Streaming's `KafkaUtils`. Nevertheless, it is important to understand the fundamentals for configuring the Kafka consumer—the streaming application—and for understanding possible errors which may be caused by Kafka's internal mechanisms.

3.2 Anomaly Detection Measures

In order to perform anomaly detection in our system (cf. Chapter 4), two measures are computed over the streams: relative entropy and Pearson correlation. In this section, these two measures are briefly explained.

3.2.1 Relative Entropy

The relative entropy, or *Kullback-Leibler Divergence* [21], $D(P||Q)$, is a non-symmetric measure of information loss. Specifically, it measures the information loss when a distribution Q is used to approximate another distribution P, where P is the "true" distribution of the observed data. It is defined on two probability distributions P and Q as follows:

$$D(P\|Q) = \sum_{i \in A} P(i) \log \frac{P(i)}{Q(i)} \tag{3.1}$$

where $P(i)$ and $Q(i)$ are the probability of item i in the respective probability distribution, given by

$$P(i) = \frac{m_i}{\sum_{a \in A} m_a} \tag{3.2}$$

where A is the set of all possible items i in the probability distributions and m_i and m_a are the number of items i and a, respectively, in the current distribution P.

Relative entropy is used to measure the difference between two probability distributions P and Q representing two datasets, for example for the purpose of detecting anomalies [23]. This is a parameter-free generic method that applies to multidimensional data without requiring any prior domain knowledge about the underlying distributions P and Q. In our context, $D(P\|Q)$ is used to measure changes between successive time windows over multi-dimensional data streams, as introduced by Dasu et al. [6].

The values of $P(i)$ and $Q(i)$ are defined over $[0, 1]$. D is not defined over a fixed range. In order to be able to interpret the value, it is therefore necessary to determine a baseline as a range of normal values for relative entropy. Under the premise that there exists a normal profile of the data, low relative entropy is linked to regularity. A low relative entropy indicates that the two distributions P and Q are similar. $D(P\|Q)$ is 0 if the distributions P and Q are identical. Anomalies are detected when the relative entropy increases, i.e., when D increases significantly compared to the baseline.

3.2.2 Pearson Correlation

The Pearson correlation coefficient is a statistical value measuring the linear dependence between two vectors X and Y, which are assumed to be normally distributed. The vectors X and Y contain n elements each, denoted $x_1 \dots x_n$ and $y_1 \dots y_n$. Pearson correlation is defined over X and Y as

$$r(X,Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{3.3}$$

where $\bar{x}$ and $\bar{y}$ stand for the mean of X and Y, respectively.

The coefficient $r(X,Y)$ ranges between 1 and -1. Positive values from $(0,1]$ indicate positive correlation between X and Y, while negative values from $[-1,0)$ indicate negative correlation. A positive $r(X,Y)$ occurs when an increase or decrease in the values in X is met with the same trend, increase or decrease, in Y. A negative $r(X,Y)$ occurs when changes in X and Y are opposing, e.g., a decrease in one vector is met with an increase in the other vector. When the Pearson correlation coefficient is 0, there is no correlation between X and Y.

It should be noted that Pearson correlation coefficients are limited in the sense that they detect strictly linearly correlated data; for other types of correlation between two datasets, Pearson correlation may yield a value of 0 indicating no correlation between the data. However, the data that are being considered for correlation (explained in detail in Section 4.2.2) are either linearly correlated or uncorrelated.

3.3 Data Sources and Types

This section clarifies the nature and the different formats of the various real-world data streams and sources, provided by Swisscom's telecommunication network and big data infrastructure, upon which anomaly detection is performed.

3.3.1 Telecommunication Network Monitoring Interfaces

Figure 3.4 describes the cellular network infrastructure, the components that are required by the different protocols (2G and 3G) and the locations at which the probes, from which we receive event streams, tap on the data flow and monitor the activity. Fundamentally, the infrastructure of the cellular networks consists of a Global System for Mobile Communications (GSM), a Radio Access Network (RAN), and a Core Network (CN). The latter is split into circuit switched (CS) (voice calls) and packet switched (PS) (data traffic) domains. Mobile devices can attach to CS or PS, or both at the same time, with a preference for newer technologies (e.g., 4G rather than 3G or 2G). The radio communication takes place between a mobile device and a base station within the current GSM or RAN, serving one or more radio cells, which then carries the voice and data traffic via fixed network links to/from CN. Radio cells are the smallest spatial entities

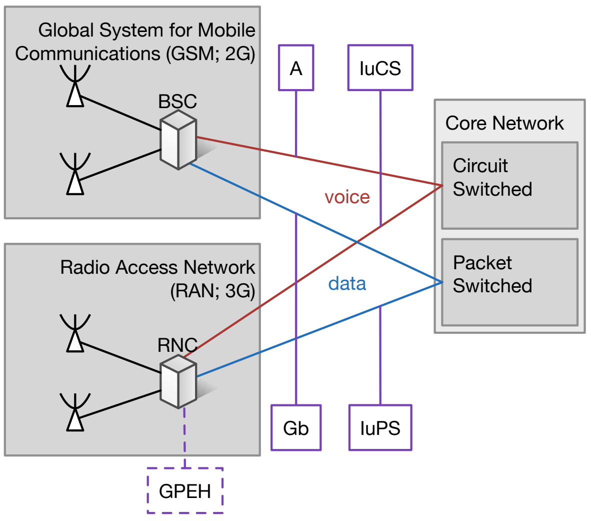

Global System for Mobile Communications (GSM; 2G)
BSC
A
IuCS
Core Network
Circuit Switched
voice
data
Packet Switched
Radio Access Network (RAN; 3G)
RNC
Gb
IuPS
GPEH

Figure 3.4: Simplified schematic overview of components of the telecommunication network and the links between them (adapted from internal wiki). Purple boxes show the monitoring probes and the links they tap on.

in the cellular network and can be classified as either 2G (GSM/EDGE), 3G (UMTS/HSPA), or 4G (LTE). For the purpose of quality assurance, a passive monitoring system collects signaling events from the links between the GSM or RAN and CN parts of the network, covering all 2G, 3G, and 4G—specifically on the A, Gb, IuPS, IuCS, and S1-MME interfaces.

While all streams carry certain network events, which are multidimensional with regards to their attributes, the attributes differ depending on the protocol and the function (voice or data). Since some applications may only be interested in events from a certain interface, each of these interfaces is provided in the form of a separate stream. The interfaces and types of data will be described in the following paragraphs.

Geo Probes capture events being transmitted between the RAN or GSM (distributed throughout Switzerland at different locations, at a coarser granularity than cells) and the centralized CN, as shown in Figure 3.4. Geo Probes provide data—monitoring information of the activities—on four interfaces, i.e., on four links in the network, also referred to as *DataCast* interfaces: *A* for 2G voice events, *Gb* for 2G data traffic events, *IuCS* for 3G voice events, and *IuPS* for 3G data traffic events. Monitoring probes on these interfaces capture the traffic between the Base Station Controller (BSC) (2G interfaces) or the Radio Network Controller (RNC) (3G interfaces), respectively, and the CN. One RNC controls multiple 3G base stations (cells) in a larger geographical area. The RNC topology in Switzerland can be seen in Figure 3.5. In the names of the UMTS interfaces IuCS and IuPS *Iu* stands for interface and the component of the CN, CS or PS, indicates the voice or the data interface.

The 4G protocol uses a different network infrastructure, not distinguishing between CS and PS, and is therefore not included in the schema. 4G events are captured by probes on the S1-Mobility Management Entity (MME) interface.

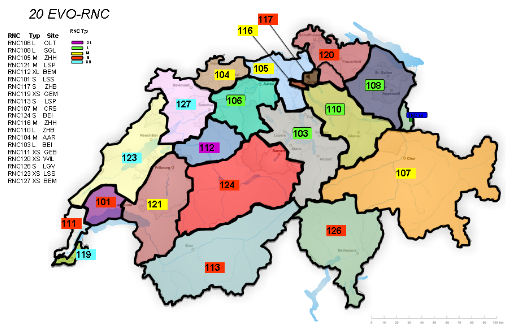


Figure 3.5: Spatial partitioning of Switzerland into RNCs. Each RNC controls a set of cells within the larger area it covers.

General Performance Event Handling (GPEH) is a source for network monitoring and diagnostics, captured at RNCs for 3G events instead of on the links between the RAN and CN. Compared to Geo Probes, the users' activity is captured at a finer granularity due to the probes' greater proximity to the user equipment. As a result, the stream is much bigger than the Geo Probe data (cf. Table 3.1).

In our setup, streams coming from these interfaces can be regarded as sequences of tuples, where one tuple is associated with one event, and an event is triggered by any of a number of different activities, including phone calls, establishing a data connection, or moving from one location to another that is covered by a different base station. The event tuples consist of multiple attributes, many of which are highly domain-specific and not relevant for the purpose of the system presented in this thesis. These attributes vary between the interfaces, but common ones include the start and end time, event type, protocol, Location Area Code (LAC)—essentially an aggregate of cells—, current cell ID, and the anonymized user ID, IMSI.

There exist a large number of distinct event types (call types for voice interfaces and transaction types for data interfaces) which are captured on the links by the probes. Depending on the probes' configuration, which in turn depends on the use case for which the data are monitored, events are collected and monitored when they are of some defined type. In this thesis, only events of types which are relevant to mobility insights are considered. This includes events relating to any activity (phone calls, text messages, data transaction) taking place at a certain location and passive events that are triggered when users move between locations (in which case a so-called *handover event* is sent to the network for the purpose of being able to route incoming calls).

Due to the high dimensionality of the tuples, the first step of processing is commonly a `map` operation in order to reduce the size of the events on the stream for further processing by keep only the necessary attributes. This dimensionality reduction can be seen as an analogy to projections when retrieving data from RDBMS. Given the cardinality of the streams, this is a reasonable measure in order to allow us to handle the streams in real time.

Table 3.1 provides an overview of the approximate cardinalities of the different streams in terms of bytes as well as number of events during daytime. The amount of events varies throughout the day. Figure 3.6 shows the periodicity of the data volume for the 2G, 3G, and

Table 3.1: Approximate stream cardinality and byte size during business hours (commonly observed values).

Interface	Events/sec	Bytes/sec
A	1500	195 KB
Gb	500	70 KB
IuCS	3900	580 KB
IuPS	3800	480 KB
S1-MME	54000	17.6 MB
GPEH	187500	23.5 MB

4G interfaces during a 24 hour period on a weekday as displayed in the existing monitoring dashboard. There is a notable drop during the night, between 22:00 and 7:00, and another slight periodic decrease after noon. Spikes in the MME stream are related to the processing of event counts and writing to InfluxDB as part of the monitoring infrastructure and do not present themselves as anomalies in the data. Overall, we can observe that the volume of the data increases for more recent generations of telecommunication protocols. The newer protocols have a significantly higher usage since devices use newer technology when possible and connect to cells belonging to older types of technology only when there is no alternative. When dealing with batch data, we have around 7 GB per day on the 2G interfaces, 18 GB per day on the 3G interfaces, and 240 GB per day on the 4G interface.

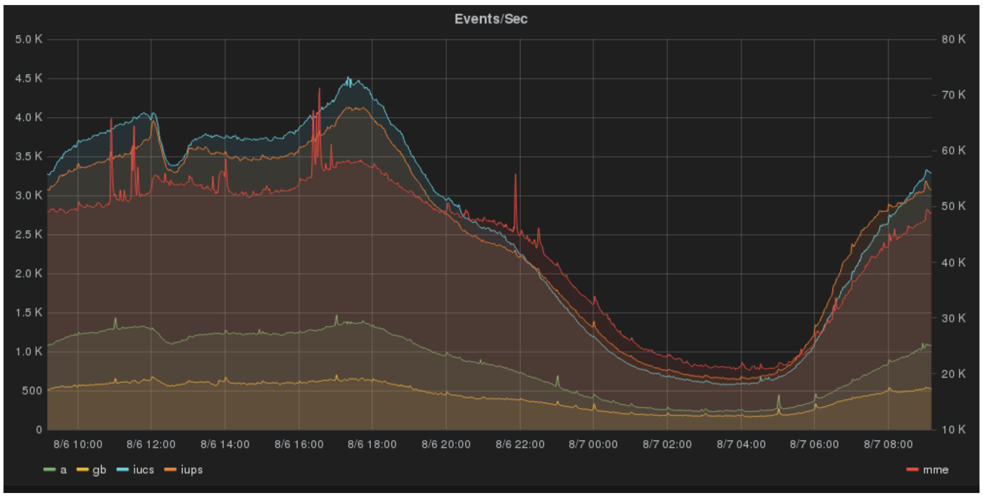


Figure 3.6: Time series monitoring dashboard showing the number of events per second on each interface throughout a 24 hour period.

3.3.2 Firehose

Firehose has recently been put into place as a big data streaming infrastructure within Swisscom. This infrastructure provides a pipeline delivering data from different raw binary sources to the application layer using Kafka (cf. Section 3.1.2).

Figure 3.7 displays the architecture and pipelines within Firehose. The raw—binary—input data are split, knowing that each event record and each field in the record consists of a fixed number of bytes. The input stream of bytes is split into event records; each event record is then split into its attributes, parsed, and converted to the common record format before being serialized and sent to the respective Kafka topic. Each Kafka topic corresponds to one of the telecommunication network monitoring interfaces. Events arrive in proper chronological order and are processed in that order by Firehose.

Kafka is particularly useful for our setting as it decouples the receiving and initial preprocessing of monitoring probes' data streams from the applications that consume and analyze the data, such that multiple applications can consume data in an isolated manner. By preserving a short history of the events, the Kafka queues within Firehose support fault-tolerance and the possibility for applications to recover after short-term failures.

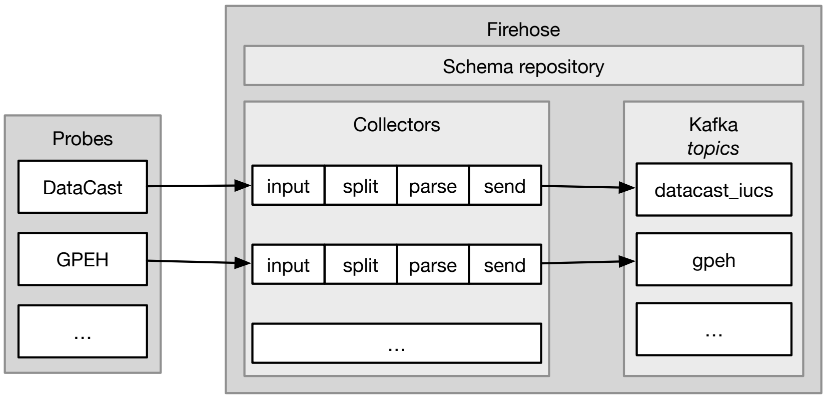


Figure 3.7: Firehose architecture (adapted from internal wiki). Firehose receives data from probes, performs preprocessing steps, and writes the data to Kafka topics.

Apache Avro[7] was chosen as common record and serialization format for its compactness and ease of integration with the Hadoop ecosystem. All topics provide streams of `Array[Byte]`, which can be decoded into Avro objects according to predefined schemata in JavaScript Object Notation (JSON) format. These schemata are provided as part of Firehose in a centralized repository. The schema repository is accessible from both Firehose as the producing application as well as from the consuming applications.

3.3.3 Classification of Anomalies

In order to construct a system for anomaly detection, it is to be defined what constitutes an anomaly. For detecting data quality issues, three different high-level categories of anomalies will be distinguished. These are the most common known sources of anomalies in the context of the previously introduced telecommunication monitoring data.

1. **Changes to the network configuration.** These anomalies are related to modifications as a result of changes in the probe configurations. As mentioned previously, these probes are vulnerable to misconfiguration and are reconfigured once the misconfiguration is detected. There is no feedback loop for these changes from the network maintenance side to the data consumer side and not all who are impacted by these changes, including departments working on the transmitted data, are notified of the changes. As a result of

[7] `https://avro.apache.org`

configuration changes, data characteristics may change and data may have to be processed differently.

2. **Failures in hardware components.** Different network components may suffer hardware defects or software bugs. Identifying these is important for network maintenance and is also the reason for setting up the diagnostic feeds. But the data are used beyond monitoring to gain insights, e.g., about users' mobility, and thus data consumers need to be aware of any failures or quality issues in the data.

3. **Changes to human behavioral patterns.** Real-live events, such as art exhibitions, music festivals, or disruptions to vehicular traffic, cause a change in users' typical behavioral patterns. The knowledge obtained from observing these events can be used to explain changes to the data characteristics and further to predict necessary network capacities for future events.

Taking these anomalies into account, we are able to adapt the specific measures (Section 3.2) for our purposes, use real-world data for known anomalous scenarios, and simulate anomalies of the defined types.

Chapter

4

Anomaly Detection System

The system we designed for DQT by means of anomaly detection [34] and its integration within the telecommunications monitoring data pipeline is depicted in Figure 4.1. In this high-level overview, we show the entire data pipeline starting from cell towers on the left-hand side to the anomaly detection results on the right-hand side. Signaling traffic is received from network probes, as seen in Figure 3.4. The colors of the data flow in Figure 4.1 represent the streams of different data types (cf. Section 3.3.1), which are received and processed separately. For the purpose of simplicity, Figure 4.1 has been limited to show one network monitoring probe for each interface only. In reality, there are multiple (physical) probes per interface type, of which the events are collected by Firehose, our data stream enabling infrastructure (cf. Section 3.3.2). As described previously, in Firehose the data are staged in real time and then written to a dedicated queue for real-time consumption. Periodically, data are also staged to HDFS for longer-term storage and processing. The anomaly detection component consumes the data (either in real time (Kafka queue) or in batch (HDFS) mode), processes them, and is to output both metrics and alerts. The various components of the architecture are described in more detail in the following.

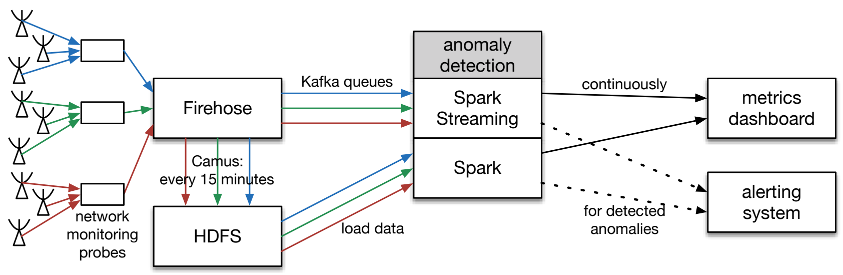


Figure 4.1: Overview of the stream architecture and the integration of the anomaly detection system therein, showing the entire pipeline from cell towers through staging and processing to anomaly detection results.

4.1 Data Collection

Signaling traffic carrying various network events is captured by probes connected to links between GSM or RAN and CN (recall the network overview and probe locations in Figure 3.4). Those captured events contain non-personal monitoring information from the mobile terminals. They do not carry any information about the content transmitted over the mobile network. There exist different types of probes and different types of data for the various network components. Incoming data is processed separately depending on the probe type. Specifically, for each generation and each network type (e.g. 2G and A or 3G and IuPS), separate probes tap into the network, producing separate data streams. Low-level, binary events are first fed to the Firehose, which parses, serializes, and writes the events to Kafka topics, one per interface. Periodically every 15 minutes, events are pipelined from the queues into HDFS through Camus[1]. Both the queues and the storage consider the same data format and can be used as input to Spark Streaming and standard Spark, respectively. As each event is timestamped, it can be treated in the same way as real-time data and can also be used to simulate streams. In that sense, our system emulates the so-called lambda architecture [27], with analogous pipelines for batch and real-time processing on the same data source.

4.2 Stream Processing System for Anomaly Detection

Our system for anomaly detection on network monitoring events is built using Spark on top of YARN. In Spark Streaming, data are processed periodically after a fixed duration as micro-batches. The duration of the micro-batches is chosen experimentally, as it depends on the volume of data. We consider several interfaces of different sizes; for example, the data volume on 3G interfaces is bigger than that on 2G interfaces (cf. Table 3.1), such that in different implementations, a different configuration may be necessary. Longer micro-batches require more storage since more data needs to be cached in-memory until the next processing interval (recall Figure 3.2, where stream data are collected for a period in time until processing is triggered for this micro-batch). This follows from the fact that all data from a micro-batch are computed jointly and hence the processing can only start once all data for one micro-batch have been received. On the other hand, a shorter micro-batch duration requires faster online algorithms and cannot amortize the network overhead from shuffling data in the same way as for longer micro-batches. Since micro-batches are computed in order, the time required to process one micro-batch, including any potential network overhead linked to the collection and repartitioning of results, cannot be longer than the duration of the micro-batch. Otherwise, micro-batches rapidly accumulate and lead to data losses (recall Figure 3.3).

Streaming receivers, which represent the input interface of the Spark Streaming component, connect as consumers to Kafka located in Firehose (see Figure 4.1 and the integration of Kafka as stream providers in Firehose in Figure 3.7). In order to load-balance the system, multiple receivers consume event streams in parallel. Nine parallel receivers are used for each stream and their partitions of the data stream are then combined for processing all data jointly. The consumers are configured to receive every event only once, beginning at the time when the

[1]`https://github.com/linkedin/camus`

application is first started. In our setup, output operations include returning a value to a network connection or a file for visualization purposes. As output from the anomaly detection component, metrics are written continuously, whereas alerts are triggered upon detection of an anomalous event.

The actual anomaly detection—in Spark and Spark Streaming, depending on the use case and data at hand—consists of computing measures over short time windows and comparing the outcome to expected values. In order to perform anomaly detection, two measures are continuously maintained over the streams: relative entropy on individual data streams and Pearson correlation across multiple streams. These metrics form a constant baseline over non-anomalous data, such that anomalous data are detected as deviations from typical baseline values.

4.2.1 Relative Entropy Pipeline

Relative entropy is computed separately on each interface by comparing the empirical distributions of event types at their respective topological level. The data pipeline for computing relative entropy is shown in Figure 4.3. As a first step, optionally, the stream is filtered to include only events originating from an area of interest. Each batch in the incoming DStream is mapped onto a new DStream of *((location, event type), 1)* key-value pairs, where the identifier for the location and the type of the event form a composite key. We consider three levels of network topology, as highlighted in Figure 4.2:

1. *cellsite*, where the location key is a 4-character identifier;
2. *regional*, e.g., per *LAC*, which is an aggregate of multiple co-located cells for larger regions within the country with a unique ID, or for specific areas of interest defined as sets of cells; and
3. *globally* for the whole country, discarding the location key.

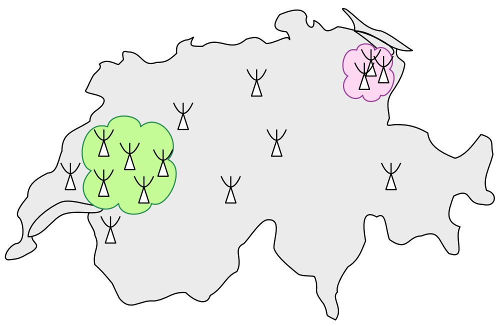


Figure 4.2: Levels of spatial granularity for aggregating data. Shown here: globally—all data for the entire country, regionally—all data from cells lying within defined areas of interest, such as the colored clouds, and locally—individually for each cellsite.

While looking at lower levels in the topology facilitates the detection of local anomalies, a more global model is faster to compute due to the smaller key space. A higher key space means for example that if we maintain one metric for each cell location, we obtain a high number of distinct metrics for each time period. If the stream is unfiltered, this number of distinct values—the total number of cellsites—is in the order of magnitude of 10000.

By summing up the values per key in a `reduce` operation, the number of events per location and event type get counted. Grouping per location yields a new RDD containing the event histograms, i.e., the counts per event type and per location.

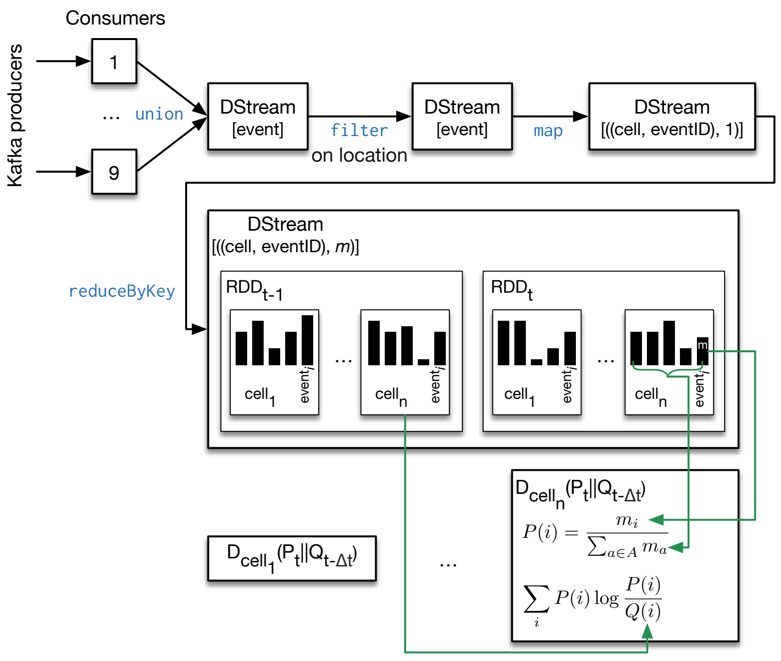


Figure 4.3: Relative entropy $D(P||Q)$ computation pipeline showing the parallel receiving of the stream, transformations, and computation of the measure from the data.

These event histograms are interpreted as aggregates of anonymized user-triggered actions since they capture various aspects of human activity (making a phone call, moving across the network, etc.). The probability $P(i)$ of each event type i in the current distribution P, as in Equation (3.1), is given by dividing the count for this event, m_i, by the sum of the counts m_a in the current histogram for all possible event types in A. Hence, $P(i)$ represents the relative frequency of event type i in the current distribution. Finally, for each location indicator, the relative entropy $D(P_t||Q_{t-\Delta t})$ between the current distribution P_t at time t and the previous distribution $Q_{t-\Delta t}$ at time $t - \Delta t$ is computed by summing the comparison of each possible event type i. A higher D than in the baseline indicates the occurrence of change. In streaming mode, the probability distribution from the previous RDD is stored for comparing adjacent windows, yielding a distance measure between the two time periods per location. We do not only compare adjacent windows but also compare the histograms of all hours, both daily and weekly. Hence, we can detect both abrupt changes and gradual changes over time. While adjacent window computation with Δt set to one hour is performed on both streaming and batch data, the current solution for comparing with larger values of Δt, such as one week, is done purely on data that is loaded from HDFS. As noted above, Spark's programming model facilitates code reuse between both real-time and batch processing and guarantees comparable results.

```
Algorithm 1: Relative Entropy
Input: $S = [[(c_1, e_1), ...], [(c_2, e_2), ...], ...]$ where $S$: stream of tuples of $c$: cell ID and $e$:
    event ID grouped into micro-batches
Output: $[[D_1, ..., D_n], [D_1, ..., D_n], ...]$ where $D_i$: relative entropy for cell $i$
begin
    stream map $((c, e), 1)$                          // composite key $(c, e)$, value 1
    foreach $[(c, e), ...]$ in $S$ do                        // for each micro-batch
        current ←∅
        previous ←∅
        foreach distinct keypair $(c_i, e_j)$ in $[(c, e), ...]$ do         // reduce by key
            current += $((c_i, e_j), \text{sum}(value_{(c_i,e_j)}))$      // count $e_j$ in $c_i$ and append
        if previous $\neq \emptyset$ then
            foreach cell $c_i$ do
                $D$ ←∅
                foreach event type $e_j$ do
                        // count of $e_j$ in $c_i$ divided by sum of all events in $c_i$
                    $P(e_j)$ ←$current(c_i, e_j)/\text{sum}(current(c_i))$
                    $Q(e_j)$ ←$previous(c_i, e_j)/\text{sum}(previous(c_i))$
                    $D$ += $P(e_j) * log(P(e_j)/Q(e_j))$        // sum for all event types
                return $D$            // relative entropy $D$ for cell $c_i$ between
                distributions at times $t$ and $t - \Delta t$
        previous ←current
```

Algorithm 1 describes, in high-level pseudo-code, the *per-cell* relative entropy computation, considering that the input stream has already been mapped from entire events containing all attributes to pairs of location identifiers—cellsites—and event types. Listing 4.1 then shows the implementation of the relative entropy stream processing in Scala using Spark Streaming, along the description in Algorithm 1 and Figure 4.3, for a *global* relative entropy value in a filtered area. For brevity, the code neither includes the parallel stream receiving and initial decoding to Avro objects nor any necessary set-up or configuration. A `map` operation removes the attributes from the events that are not required, keeping only the cell ID and event ID for each item on the stream, given as input to the algorithm. The incoming stream is filtered, mapped, reduced (lines 1–3). Then, for each RDD in the DStream, relative entropy is computed between the previous histogram and the current histogram by first computing the probabilities for each event and then summing as in Equation (3.1) (lines 10–12). The current histogram is stored in memory for the next processing interval. Here, the results are printed to the console once per micro-batch interval.

We now give a simplified example of how relative entropy is computed using sample data.

Example 1 We consider streams of messages of the form

$$[(t_1, c_1, Eid_1), (t_2, c_2, Eid_2), \dots]$$

with c_i coming from the set of cells $\{B, C\}$ and Eid_i coming from the set of possible event types $A = \{1, 2\}$. A micro-batch

$$[(t_0, C, 1), (t_1, C, 2), (t_2, C, 1), (t_3, B, 2), (t_4, C, 1), (t_5, C, 1)]$$

```scala
stream.filter(event => event.get("current_cell_lac").asInstanceOf[Int]==2020).
  map(event => (event.get("call_type").asInstanceOf[Int], 1L)).
  reduceByKey(_ + _).
  foreachRDD { rdd =>
    val current = rdd.collect.toMap
    val currentSum = current.foldLeft(0.0)(_ + _._2)
    if (previous != null) {
      val previousSum = previous.foldLeft(0.0)(_ + _._2)
      val dcp = AEvents.foldLeft(0.0){ (d, eventId) =>
        val pce = (current.getOrElse(eventId, 0).toString.toInt + 0.5)/(currentSum +
            AEvents.size/2.0)
        val ppe = (previous.getOrElse(eventId, 0).toString.toInt + 0.5)/(previousSum +
            AEvents.size/2.0)
        d + pce * log(pce/ppe)
      }
      println(dcp) // alternatively: write to file or to a network connection
    }
    previous = current.toMap
  }
```

Listing 4.1: Relative entropy pipeline in Scala using Spark Streaming.

is obtained at a time t, with timestamps t_i ranging between $t - \Delta t$, the previous micro-batch computation time, and t, the time at which the data collection ends and the computation for this micro-batch is triggered. The partition of the stream at time t is mapped onto

$$[((c_1, Eid_1), 1), ((c_2, Eid_2), 1), \dots].$$

We apply a `reduce` operation on the composite key consisting of the cell and the event type to transform this stream into tuples containing the key and the corresponding count of events in the current stream as follows:

$$[((C, 1), 4), ((C, 2), 1), ((B, 2), 1)].$$

Since we compute the relative entropy for each cell individually, we illustrate the computation for cell C only (similar computations are applied to all other cells). At time t, the histogram's counts of cell C in our example are respectively 4 for event type 1 and 1 for event type 2. Using Equation (3.2) the probabilities in P_t are respectively $P(1) = \frac{4}{5}$ and $P(2) = \frac{1}{5}$. We compare the distribution P_t to the probability distribution from a previous micro-batch $Q_{t-\Delta t}$ from cell C with $Q(1) = \frac{2}{3}$ and $Q(2) = \frac{1}{3}$. By applying Equation (3.1), we get for our example

$$D(P\|Q) = \frac{4}{5} \log \frac{4/5}{2/3} + \frac{1}{5} \log \frac{1/5}{1/3} = 0.044.$$

4.2.2 Pearson Correlation Pipeline

In order to compute the Pearson correlation coefficient $r(X, Y)$ between vectors X and Y obtained from windows at time t over two streams S_X and S_Y, the implementation consumes events from at least two separate interfaces. Both streams are treated separately, mapping each stream onto a DStream containing anonymized user IDs and then counting the number of distinct IDs per micro-batch such that we obtain one count per RDD. A graphical representation of the data transformation process for the Pearson correlation pipeline is shown in Figure 4.4. Since we cannot compute the correlation coefficients directly between two unbounded streams, we opt for windowing over the stream in order to create finite vectors, between which we are able to compute the Pearson correlation as per the definition in Equation (3.3). Windowing over the counts of micro-batches over longer durations yields RDDs containing multiple counts—essentially DStreams containing, as RDDs, the vectors X (on the windowed stream S_X) and Y (on the windowed stream S_Y). At this point, two previously separate DStreams are combined as a DStream of pairs of RDDs, (X, Y), one from each DStream, with corresponding timestamps. Using the pairs of RDDs, containing the unique input counts $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ respectively, a correlation coefficient for the particular time period t is computed according to Equation (3.3).

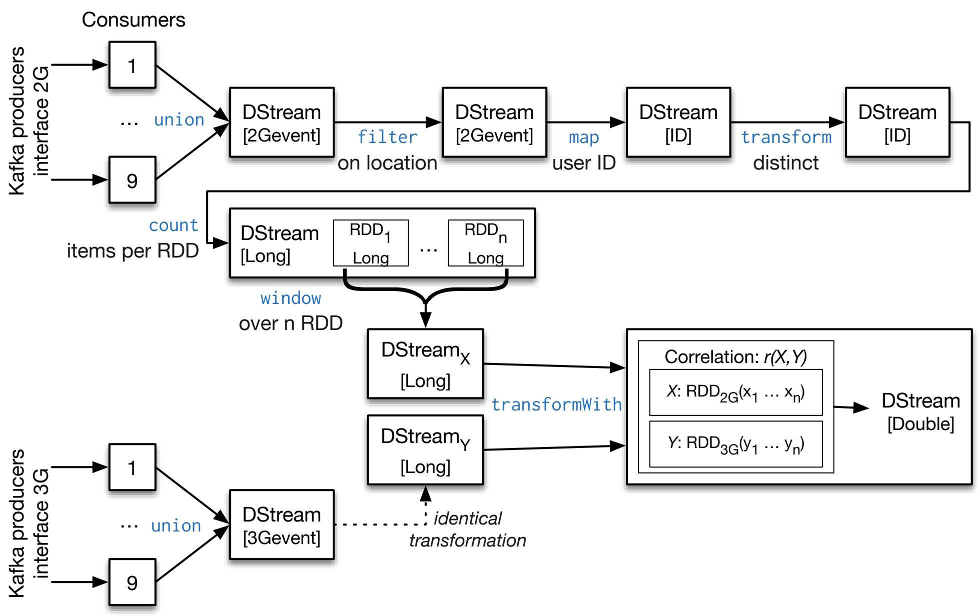


Figure 4.4: Pipeline for computing the Pearson correlation $r(X, Y)$ between windows containing the vectors X and Y over two streams. Two streams are received and transformed separately, then joined to compute the correlation between corresponding windows at the same time.

Algorithm 2 describes the implementation of the previously described pipeline for computing the correlation between windows over two streams in pseudo-code. As in the previous section, for brevity, we take as input to the algorithm the received and preprocessed stream, where all attributes except for the IMSI, the anonymized user ID, have been removed. Since we leverage the Pearson correlation function available from Spark, we do not elaborate on the computation of Equation (3.3) in the algorithm. The implementation in Scala and Spark Streaming is then shown in Listing 4.2 for global correlation between the streams transmitting events from the A and IuCS interfaces. Filtering the raw input events to process only those from one LAC is performed on line 2. On lines 3–5, we obtain the anonymized user ID and count the distinct numbers on lines 6–7. By windowing over the stream every fixed `windowDuration`, a DStream of RDDs of multiple counts is obtained. The same processing is performed on the second stream (line 10). A `transformWith` operation joins the streams and computes the Pearson correlation between vectors containing counts for both streams, using the function provided by Spark's machine learning library, in lines 12–17.

Algorithm 2: Pearson Correlation Coefficients

```
Input: $S_X = [u_1, u_2, \ldots]$, $S_Y = [u_1, u_2, \ldots]$ where $u$: anonymized user ID, $S_X$ and $S_Y$:
       continuous, unbounded streams
Output: $[r(X,Y), r(X,Y), \ldots]$ where $r(X,Y)$: Pearson correlation between windows
        over the streams
begin
    $RS_{X,Y} \leftarrow \emptyset$                    // result stream containing counts per batches
    foreach batch from $t_v$-$t_w$ in the streams $S_X, S_Y$ do // $t_w - t_v$ corresponds to the
    length of one batch, starting at $t_v$ and ending at $t_w$
        $x_{t_v} \leftarrow$count(distinct($S_{X\,t_v - t_w}$))
        $y_{t_v} \leftarrow$count(distinct($S_{Y\,t_v - t_w}$))
        $RS_{X,Y}$ += $(x_{t_v}, y_{t_v})$          // append pairs of counts from both streams
    foreach window from $t_q$-$t_p$ over $RS_{X,Y}$ do   // duration of $t_q$-$t_p$ is a multiple
    of the duration of $t_v$-$t_w$
        $X \leftarrow (x_{t_q}, ..., x_{t_p})$
        $Y \leftarrow (y_{t_q}, ..., y_{t_p})$
        $r_{t_p}(X,Y) \leftarrow$pearsonCorr($X$, $Y$)  // compute correlation using the provided
        function
        return $r_{t_p}(X,Y)$
```

```scala
1   val streamIucs: DStream[Long] = rawStreamIucs.
      filter(_.get("lac").asInstanceOf[Int]==2020).
3     map(event => event.get("imsi").asInstanceOf[ByteBuffer]).
      filter(imsi => imsi != null).
5     map(imsi => new String(imsi.array()).toLong).
      transform(rdd => rdd.distinct).
7     count().
      window(Seconds(windowDuration), Seconds(windowDuration))
9
    val streamA: DStream[Long] = rawStreamA.// [...]: identical pipeline
11
    streamIucs.transformWith(streamA, (rddIucs: RDD[Long], rddA: RDD[Long]) => {
13    Statistics.corr(
        rddIucs.map(count => count.toDouble),
15      rddA.map(count => count.toDouble),
        "pearson")
17  })
```

Listing 4.2: Pearson correlation pipeline in Scala using Spark Streaming.

A simple example on sample data illustrates the pipeline.

Example 2 Given two distinct streams of messages mapped onto a stream of anonymized user identifiers Uid_i of the following form

$$[Uid_1, \ldots, Uid_n],$$

we collect the data during a short period of time (e.g., 10 seconds), then count the number of distinct IDs during this period and write the counts to a new stream.

Let us assume stream S_X contains

$$[A, B, B, A, B]$$

and stream S_Y contains

$$[C, B, C, C, D, C, D, E, F, A]$$

in this short window. By applying a `distinct` operation on each stream (for S_X, yielding $[A, B]$, and for S_Y, yielding $[A, B, C, D, E, F]$) and then retrieving the length of this window, we obtain the count of distinct users per timeframe. These two counts, respectively 2 and 6, are then written to the respective output (result) streams RS_X and RS_Y. After 40 seconds, i.e., after receiving three more micro-batches on each stream yielding further counts, the previous output streams contain: RS_X,

$$[2, 1, 1, 3],$$

and RS_Y

$$[6, 5, 4, 6].$$

By windowing over these counts (e.g., every 40 seconds) over both streams, we obtain two vectors X and Y from the same timespan, each containing, in this case, 4 counts. Grouping these vectors into pairs of (x_i, y_i) gives, as $RS_{X,Y}$,

$$[(2, 6), (1, 5), (1, 4), (3, 6)].$$

Plugging the values into Equation (3.3) yields a Pearson correlation score of

$$\frac{9}{\sqrt{55}} = 0.94.$$

Consider the case where the network monitoring probe producing events on S_Y fails, such that we no longer receive events from one area. Then, by reducing the events on S_Y and the count of distinct users on RS_Y at a certain time, e.g., after 20 seconds, an increase in x_i meets a decrease in y_i. Thus, the stream of grouped pairs is as follows:

$$[(2, 6), (1, 5), (1, 2), (3, 5)]$$

so that the correlation $r(X, Y)$ for this pair of vectors decreases to

$$\frac{5}{3\sqrt{11}} = 0.5.$$

Chapter

5

Empirical Evaluation of the Anomaly Detection System

To evaluate the efficiency and the effectiveness of our anomaly detection pipeline, we conducted experiments on real-world big data streams, as well as data storage (HDFS). Both data sources are provided by Swisscom's big data infrastructure. The data we focused on for our experiments are captured at the A and the IuCS interfaces by the probes monitoring 2G voice and 3G voice links respectively, which report network events on the telecommunication network. The approximate cardinality of the streams was given in Table 3.1.

We are interested in evaluating the system both in terms of its accuracy—how well it detects anomalies (Section 5.1)—and its scalability—how well it can adapt to both an increase in computing parallelism and an increase in data load (Section 5.2).

The experiments have been performed on a development cluster composed of 24 machines with 2.1 TB of available memory in total. As a platform providing the core of Hadoop, we are using the Hortonworks Data Platform (HDP) 2.2 with YARN as the data operating system as a link between HDFS and the applications. We use Spark 1.3.1 and receive streams from Kafka 0.8.2.

5.1 Anomaly Detection Accuracy

In the following, we evaluate the accuracy of our anomaly detection for two types of events out of the three known causes for anomalies described in Section 3.3.3. First, we evaluate the detection accuracy for two real-world events where anomalies from human behavioral patterns can be observed, caused for example by sports events, popular concerts, exhibitions or vehicular traffic disruptions. For this type of anomaly, data are available for known events. Secondly, we evaluate the detection accuracy for the case of failing IT infrastructure components, for which we simulate realistic anomalous data. We apply our solution to these data in order to assess how well the known anomalies are detected.

5.1.1 Relative Entropy Accuracy

Anomalous Human Behavior

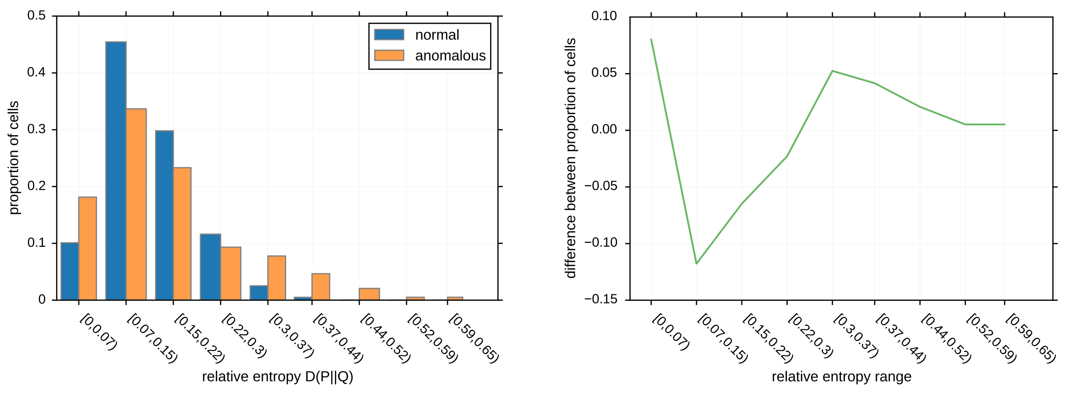
As explained previously in Section 4.2.1, the event histograms are interpreted as an aggregate of anonymized mobile phone users' non-personal activity within the network. In the experiments in this section, each histogram is constructed over the counts of events per event type, as outlined in the example in Section 4.2.1. Under non-anomalous circumstances, human behavior is mostly regular, i.e., there is no major change in the relative proportion of the counts for each event type in different histograms for a sufficiently large population of mobile devices. However, large-scale anomalous events, which disrupt movement patterns and lead to sudden changes lasting over an extended period in time, cause a change in the distribution of the events. For example, the proportion of events of the type "phone call" may increase in a situation where vehicular traffic is blocked whereas the proportion of events of the type "moved to another cell" may decrease at the same time. By measuring the difference between usual—baseline—and observed histograms, it is possible to detect change in the habitual patterns using relative entropy.

Flood in Geneva As a real-world example of an anomaly relating to a human event, we consider the flood that took place in Geneva on May 2, 2015 as a result of heavy rain. The data for this day are available from HDFS. This event caused a significant change in the movement patterns of telecommunication network users as several bridges had to be closed and users had to pick new routes to get to their usual destinations. The change in behavioral patterns implies a higher relative entropy. When considering the distribution of the relative entropy, the anomaly could potentially be detected through the unusual proportion of higher relative entropy values in the anomalous scenario (on May 2) compared to the baseline scenario. Other days with no known major behavior or movement disruption were used as the baseline.

In this first experiment, the relative entropy $D(P\|Q)$ is computed on a *per-cell* level between histograms of adjacent windows with a duration of one hour per window. The two distributions P and Q which are being compared are hence

- P_t, i.e., the distribution of the event types in all events that occurred during the one hour period before the time t; and
- $Q_{t-\Delta t}$, where Δt is one hour, i.e., the distribution of the event types occurring between two hours and one hour before t.

For every cell within the area of interest, a single value of $D(P\|Q)$ is computed for every hour of the day. We filtered the incoming stream to events originating from cells within the city of Geneva. Figure 5.1(a) shows the distribution of the mean relative entropies per cell. For every cell in the area of interest, the mean was computed over all hourly relative entropy values during one day. For the baseline, the mean entropy $\bar{D}$ over all cells is computed and the ranges for the bins on the x-axis are taken in relation to $\bar{D}$, where $\bar{D} \approx 0.15$. The y-axis shows the relative proportion of cells with mean relative entropy falling into the range given on the x-axis. For comparison, we are differentiating between the data for the known anomalous day and a baseline average obtained from computing the relative entropy on multiple days in April and May.



(a) Distribution of the proportion of cells where the mean relative entropy falls into the respective range, for the baseline and the anomalous day.

(b) Difference between the proportion of anomalous cells' means and the proportion of baseline cells' means per range.

Figure 5.1: Distribution of cells' mean relative entropy between adjacent windows throughout one day. Baseline from multiple normal days compared to the anomalous scenario on May 2, 2015.

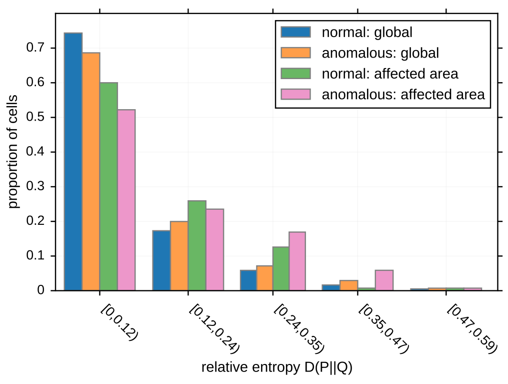
The results show that the majority of cells display lower relative entropies for both normal and anomalous days, although normal days' entropy values are more strongly located around $\bar{D}$ with lower variance. On the other hand, the proportion of cells where the mean daily relative entropy exceeds $2\bar{D}$, i.e., bins with ranges greater than 0.3, is clearly higher on May 2 than on average on known normal days. Figure 5.1(b) compares the proportions of cells for each range between the anomalous day and the baseline. The figure supports the previous observation: We see fewer cells with D within the lower ranges $[0.07, 0.3)$ on the anomalous day than in the baseline, but for the higher ranges, starting at a mean relative entropy of 0.3, we observe an increase in D for the anomalous day. This indicates that maintaining the relative entropy and counting the number of cells exceeding a threshold is a suitable way to detect anomalies in the form of human behavior changes. The higher proportion of cells with low relative entropy in $[0, 0.07)$ could be explained by the fact that we may obtain very low relative entropy values for cells where there is no activity. That is, if the traffic disruption makes certain areas covered by some cells unreachable, we would not be seeing any activity and therefore obtain atypically low relative entropy values.

By comparing days that are known to be normal to days that are known to be anomalous, we can establish thresholds for alerting about anomalies. These thresholds are set for the number of cells within the geographical area where the relative entropy value exceeds another threshold. We consider a threshold relative to the baseline mean as $k\bar{D}$ and count the number of cells where the threshold is exceeded. In our deployment, setting k to 2 gives the best experimental results. For example, in the above scenario in the city of Geneva, the number of cells with mean relative entropy above a threshold $k\bar{D}$ exceeds the number of cells above this threshold on a normal day by $m\sigma$, where σ is the standard deviation to the count of cells with relative entropy exceeding the threshold on normal days. In a series of experiments on this scenario, setting m to 1.5 yielded an accurate detection of the anomalous event.

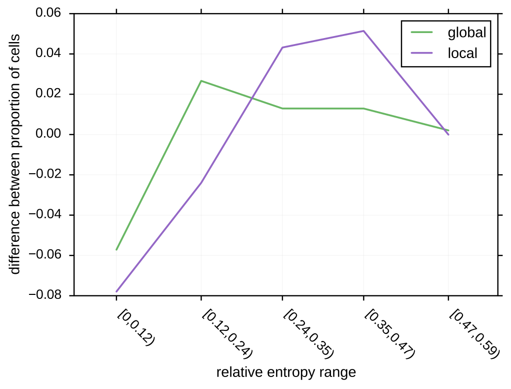
Fire at the Lausanne train station In the experiment of Figure 5.2, we consider another event, where a fire in the Lausanne train station interrupted the entire rail traffic in Western Switzerland on the afternoon of June 22, 2015. Compared to the previous event, which affected only the Geneva area, the region which was affected by this interruption is much larger. We use the dataset from the entire day, which is available from HDFS, and compute the relative entropy between the hourly event histograms from June 22 and from a non-anomalous day, as well as between two non-anomalous days as a baseline for comparison.

For comparing two days day_1 and day_2, we compute the relative entropy $D(P\|Q)$ between the histogram P summarizing hour h in cell c on day_1 and the histogram Q summarizing the same hour h in the same cell c on day_2. This allows us to take into account periodical behavioral changes throughout the day and compute the relative entropy between two histograms that are expected to be similar.

We performed such computations on two geographical scales. First, the relative entropy is computed per cell *globally* for all data available in Switzerland. Small-scale events affecting few cells (such as the previous scenario) are typically not observable on the global scale, but only on the local scale. Second, the data were filtered to include only cells from within the city of Lausanne, allowing us to observe the impact of the event *locally*.



(a) Distribution of the proportion of cells where the mean relative entropy falls into the respective range, comparing globally and locally between two normal days, as well as a normal and an anomalous day.



(b) Difference between the proportion of anomalous cells' means and the proportion of baseline cells' means per range, globally and locally.

Figure 5.2: Distribution of cells' mean relative entropy between the same hours on different days, comparing anomalous events from June 22, 2015 with a baseline, both on a global and on a regional scale.

Similarly to the experiment of Figure 5.1(a), the results are plotted on a histogram showing the distribution of the cells' mean relative entropy (averaged over all hourly relative entropy values for each cell) in Figure 5.2(a). In the global relative entropy distribution, we can observe a higher proportion of cells with higher mean relative entropy in the anomalous case than in the normal case. Where $D(P\|Q)$ exceeds $2\bar{D}$—with $\bar{D}$ being 0.12 here—the orange bars (proportion of anomalous cells, globally) are bigger than the blue bars (proportion of normal cells, globally) and the pink bars (proportion of anomalous cells, locally) are bigger than the green bars (proportion of normal cells, locally). The differences between the proportions of cells for each range are further shown in Figure 5.2(b). We observe that there is an increase in the proportion of cells with higher D on the anomalous day both on a global and on a local scale. The differences between the normal and the anomalous metrics are more pronounced when considering only the area of Lausanne, which is affected the most by the event.

The results of Figure 5.2 show that large-scale events affecting a major area and hence many cells are detectable on a global level but are more pronounced on a local level. When using thresholds for anomaly detection, the parameters need to be defined individually for different geographical scales since the global impact will commonly be lower than the regional impact.

5.1.2 Pearson Correlation Accuracy

The physical network monitoring probes from which we obtain the data are located close to Swisscom's CN. There is no one-to-one link between physical monitoring probes and cells, as each physical device is responsible for an aggregate of cells. Probe failures are therefore detectable by looking at large-scale changes, which can be observed by maintaining global Pearson correlation coefficients.

Since none of the probes recently failed in our network, we resort to simulations that aim to imitate realistic failure scenarios of network monitoring probes. We simulate two types of failure scenarios: *hardware failures*, where one probe ceases to transmit events for the area it monitors, and *software failures*, as the result of misconfiguration, where a gradually increasing duplication of transmitted events takes place.

Maintaining coefficients at a global scale is more efficient in terms of computation and storage space since fewer distinct instances of the metric (one global metric for each timeframe, as opposed to many local metrics—e.g., one for each cell—per timeframe) have to be maintained. Nevertheless, due to the high computational cost of simultaneously processing several streams, this simulation was executed on events coming from only one LAC. This is a downscaled failure scenario of the case where the hardware or software of one monitoring component belonging to a specific LAC on one interface is defective; instead of processing all data and modifying the data from one LAC, we are modifying a fraction of the events being transmitted from this LAC.

In the non-anomalous scenario, the data streams coming from different telecommunication interfaces (2G and 3G, specifically) are highly correlated in the counts of users on the interfaces during a period in time. This domain knowledge helps to maintain correlation coefficients in order to detect changes that affect components belonging to one of the two interfaces.

Abrupt Infrastructure Failure

Hardware infrastructure failures typically yield to abrupt changes. In our case, fewer events get transmitted from the respective monitoring probe which leads to lower user (i.e., input) counts since no users are counted for the area in which the probe has failed. For simulation, we filter out a proportion of the received events after a certain time. In such a case, a sudden drop in the Pearson correlation $r(X, Y)$ occurs. Due to a mostly uniform loss of events, subsequent windows again have higher correlation, hence we detect the anomaly based on the abrupt drop of events. Within the currently monitored window of the counts over the stream, the position at which the failure happens impacts the correlation score. If the failure happens precisely between two windows, no decrease in $r(X, Y)$ is observed. We therefore use overlapping windows to optimally capture every possible failure.

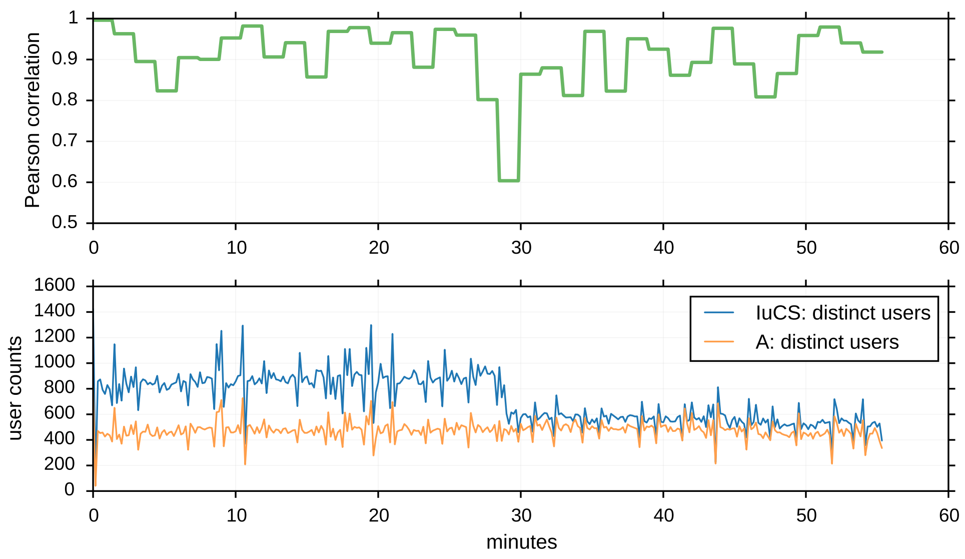


Figure 5.3: Simulating the impact of the cessation of data transmission from one probe, i.e., losing a fraction of the events on one interface, on the global correlation between the A and the IuCS stream. The correlation is computed in terms of the number of distinct users that are active within the given time window.

Figure 5.3 displays the results of computing the Pearson correlation $r(X, Y)$ between windows over the counts of distinct users on the 2G voice (A) and the 3G voice (IuCS) streams during one hour, with one count every 10 seconds and one correlation score every 90 seconds. X and Y are vectors, each containing 9 counts, corresponding to the counts in one 90 second window on the stream originating from the A and IuCS interface respectively. After 30 minutes, we filter out one third of the IuCS stream counts.

The results show that both before and after the failure, the Pearson correlation between the counts over the two streams is consistently high (ranging between 0.8 and 1). Before the failure, high correlation between the streams is our baseline for detecting outliers. We say the streams are highly correlated when the correlation coefficients are greater than 0.8. At failure time, there is a momentary decrease of the correlation to 0.6 during one 90 second window, after which the Pearson correlation stabilizes to the previous range. Because the event loss is uniform, the correlation remains high even when parts of the stream are lost, but the score is impacted at the time of change. This momentary decrease of 0.3 is significant considering the baseline's mean of 0.91 having a standard deviation σ of 0.06. We detect anomalous cases by identifying correlation coefficients that deviate from the average by $k\sigma$; in our deployment, picking k to be 4 yields an accurate detection of infrastructure failures in time.

Gradual Infrastructure Failure

Failures to the infrastructure components can not only occur in hardware but also in the software running on monitoring probes, of which there have been occurrences in the past. As configuration has to be done manually by humans, the software is vulnerable to misconfiguration.

In previous real-world failure cases, events have been transmitted multiple times, i.e., duplication occurred. The amount of duplication increased gradually over time. These duplications are hard to detect due to their gradual nature; it may take up to two weeks to spot the issue using current monitoring techniques such as counting. Therefore, we evaluate Pearson correlation as a possible means of detecting gradual duplication faster.

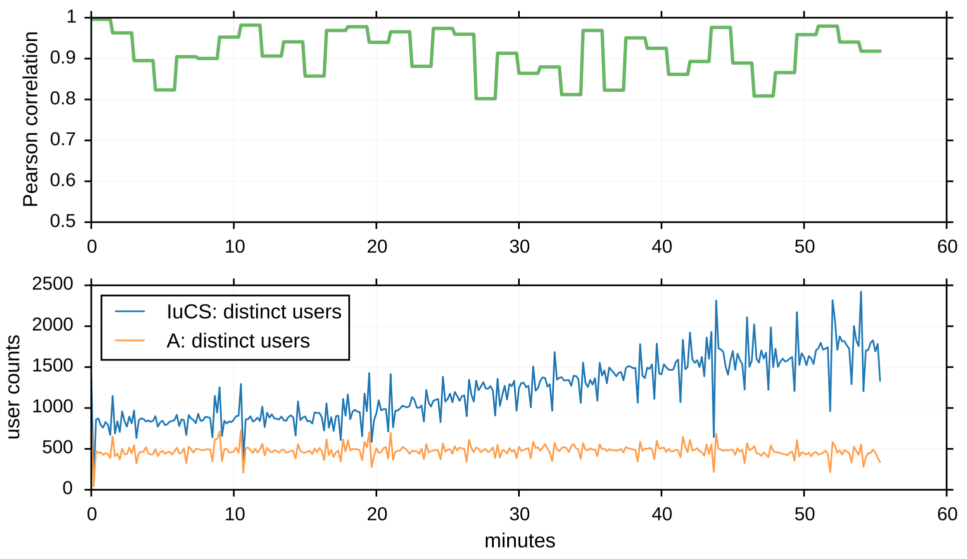


Figure 5.4: Simulating the impact of a gradual increase in event duplication as an increase in the distinct user counts on the global correlation between the A and the IuCS stream in terms of the number of distinct users at the given time.

The simulation of gradual increase in distinct user counts (for example as a result of re-emission of previous events) has been achieved in a similar manner as previously the abrupt failure. After a short period, the counts on the IuCS interface were increased gradually over time.

Figure 5.4 shows the counts as well as the Pearson correlation coefficients. We observe that, although the counts on the IuCS interface increase greatly (significantly faster than in a realistic scenario), the correlation consistently remains within the range that we consider highly correlated (greater than 0.8).

Anomalous Human Behavior

As in Section 5.1.1, we evaluate the accuracy of the anomaly detection approach using Pearson correlation for detecting anomalous events caused by changes to human behavior, specifically, for the scenario where the railway traffic was interrupted after a fire at the Lausanne train station. Unlike the previous evaluation of the Pearson correlation, we use batch data in this experiment since we need to access the historical data from the A and IuCS interfaces stored in HDFS for the anomalous day. The implementation is configured such that one count of distinct users is emitted for every 10 minute window and using these counts, one correlation coefficient is computed per hour of the day.

Figure 5.5 shows the Pearson correlation throughout the day for two normal days and the anomalous day. While there are some general trends that can be observed, overall, there is no

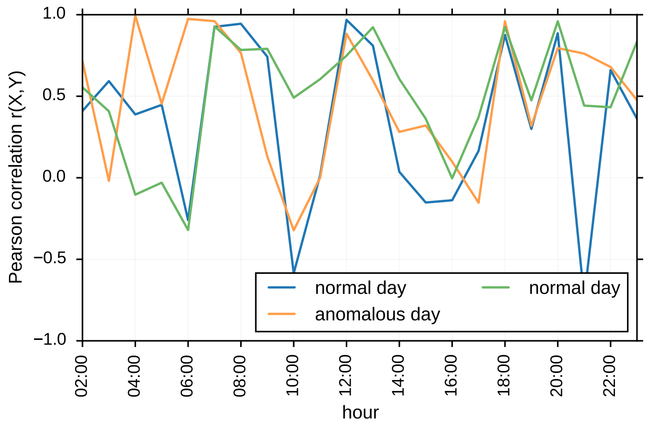


Figure 5.5: Detecting human events through Pearson correlation coefficients on batch data, where the correlation is computed between longer temporal windows with counts per 10 minutes and a correlation value per hour. At this temporal scale and for this event, the counts are not correlated.

notable difference between the normal days and the anomalous day in particular around the onset of the event, i.e., during the afternoon hours. Furthermore, comparing to the previous experiments in this section, we do not observe the highly correlated baseline; instead, the hourly correlation values, although mostly positive, cover almost the entire range of $[-1, 1]$, indicating that the counts over longer time periods are not necessarily as highly correlated as when considering short partitions of the stream. We conclude that Pearson correlation as a measure for detecting anomalous real-world events does not work well in batch mode when computing coefficients at a larger temporal scale (compare hourly windows over 10 minute counts to 90 second windows over 10 second counts, as in the previous experiments). Further, events related to changes in human behavior do not affect the correlation between the user counts on two (or more) interfaces since, in the case of an increase in human activity, all network protocols—2G, 3G, and 4G—are affected at the same scale.

5.1.3 Comparison to Anomaly Detection Techniques

In the following, we apply state-of-the-art anomaly detection techniques [29], [41] to our data and the specific anomaly detection scenarios—real-world events—in order to evaluate their applicability.

Volume of Telecommunication Activity

One approach to detecting local anomalies over telecommunication data is through quantifying the volume of activity on a local scale [41]. This is similar to the existing count-based monitoring dashboard (cf. Figure 3.6), but at a finer spatial granularity. We count the number of events per 30 minute window in the previously used scenario where there was a fire at the Lausanne train station (cf. Section 5.1.1) and compare the counts to average counts on normal days in Figure 5.7.

For monitoring on a per-cell level, two base station locations are considered: one at the Lausanne train station and another at the olympic stadium, located approximately 2 km from the train station. The area around the latter cell is not an important point for railway traffic.

Figure 5.6 shows part of a map of the city of Lausanne, with the approximate locations of the two considered cells. These being UMTS cells, we use data from the IuCS interface, i.e., the 3G voice interface, which is used for phone calls.

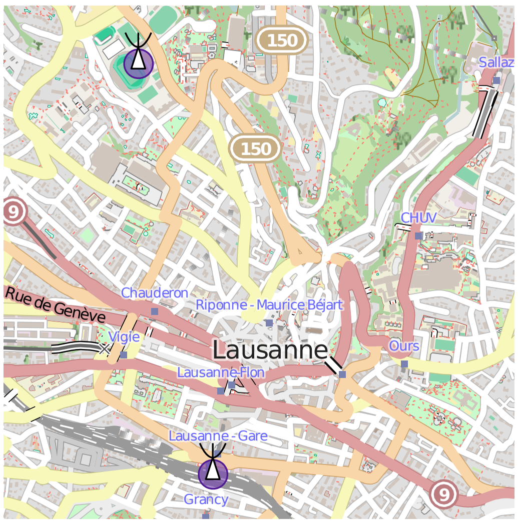


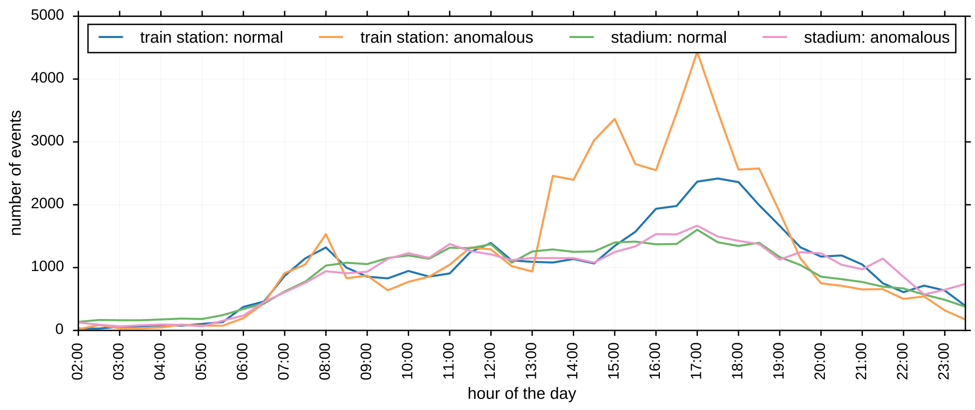
Figure 5.6: Map[1] of the center of Lausanne showing relevant cells' locations. The approximate sites of the two considered cells—at the train station (southern part of the map) and at the olympic stadium (northern part of the map)—are shown in purple.

It can be observed in Figure 5.7(a) that there is an increase in activity in the afternoon of the anomalous day around the onset of the event but no anomalous activity at the nearby location. For comparison, Figure 5.7(b) shows the event counts for the entire city of Lausanne for an average of normal days and the same anomalous day on the A and IuCS interfaces. There is no observable difference between the normal and the anomalous day at this scale. From these results we conclude that the event is only recognizable by counting at a very fine granularity. In order to continuously detect anomalies, the count measure would have to be maintained at a per-cell level. This, however, is costly in terms of memory and the number of distinct computations considering the large number of cells in the entire network area.

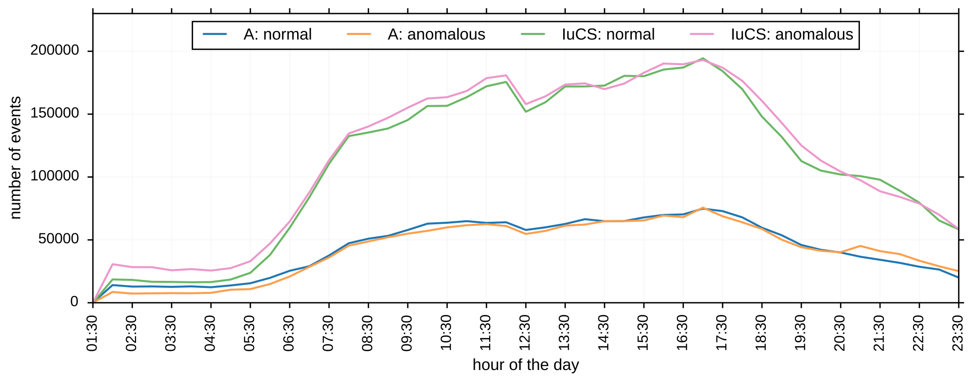
k-Means Clustering

As clustering is a common method for anomaly detection [29], we validate the accuracy of our system against an approach using k-means. For this evaluation, we load the dataset for the interruption of railway traffic in Western Switzerland after a fire at the Lausanne train station and filter out events from outside the city of Lausanne. In order to be comparable to our previously described system, we summarize temporal windows of data by computing features which are similar to the stream summaries used in computing relative entropy and Pearson correlation. These features include the proportion of each event type on the A and IuCS interface and the counts of distinct users on the two interfaces. In order to prevent the distinct user count variable from having too much weight in the k-means model, all features are normalized to the same order of magnitude. The proportions of the event types naturally range

[1]Map © OpenStreetMap contributors, `http://www.openstreetmap.org/copyright`



(a) Activity around two IuCS base stations in Lausanne: train station and olympic stadium.



(b) Activity on the A and IuCS interfaces in the entire area of the city of Lausanne.

Figure 5.7: Count of events per 30 minutes, per-cell and regional. Comparison between a baseline for normal days and an anomalous day.

between 0 and 1. The distinct user counts are divided by 10 000 (A) and 15 000 (IuCS), which are approximate observed upper bounds for the distinct user counts in a 10 minute window on a non-anomalous day. As a result, (most) values are in the same range of $[0, 1]$. This yields a total of 15 features (eight distinct event types in A, five distinct event types in IuCS, and two normalized distinct user counts) for each non-overlapping window of 10 minutes.

Spark provides implementations of the k-means algorithm for both batch and streaming mode. Since we include known anomalous data from the same scenarios as previously in Section 5.1.1, this validation is performed on batch data.

Choosing the best number of clusters, labeled k, is a non-trivial task and often ambiguous. Feature vectors are clustered according to different characteristics at different times. For example, on a normal day, we may observe a number of different clusters related to the different volume and distribution of activity during daytime, business hours, and nighttime. On an anomalous day, we may have separate clusters related to the activity during anomalous periods. A simple binary classification into normal and anomalous is therefore insufficient due to the fluctuations throughout the day. A common way of choosing k is the so-called *elbow method* [39]. According to this method, k should be chosen such that adding another cluster does not add information, i.e., such that increasing k does not decrease the within-cluster sum of squares

(WCSS). The WCSS is the sum of squared distances of all points to the nearest cluster center, which should be low if all data instances are located closely to the centroid. k is set to 7 by empirically evaluating the WCSS as shown in Figure 5.8 and choosing a first elbow point (highlighted in red).

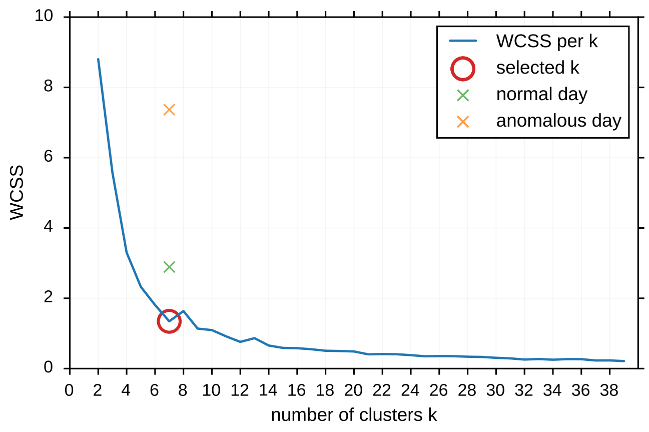


Figure 5.8: WCSS by number of clusters, choosing the ideal k by detecting a so-called elbow in the WCSS. Crosses show the WCSS when fitting normal and anomalous data to a model built on normal data using a fixed number of 7 cluster centroids.

A k-means model was trained on a normal dataset from June 30 and the WCSS values were evaluated when fitting data from another normal day to this model as well as when fitting data from another anomalous day to the model. Figure 5.8 shows these WCSSs. The WCSS curve in Figure 5.8 was estimated on the same data the model was built on, hence the WCSS is lower than when evaluating the WCSS using a different dataset. Nevertheless, it can be observed that when using the model to cluster data from another normal and another anomalous day, the WCSS is significantly higher in the anomalous dataset, as feature vectors related to the anomalies have a higher distance to the existing clusters centroids.

For the purpose of analyzing the clusters in a visual manner, the high-dimensional space (15 features per vector) is reduced to three dimensions. To achieve this reduction, the entropy within the values for each feature for all clusters is computed and those features with high entropy are chosen as the most distinctive ones for visualizing the differences. For example, one feature had a value of 0 in all cluster centroids and thus has an entropy of 0, making it not interesting as a discriminating feature. It may be removed entirely for the clustering process in order to speed up the model construction. What this entropy computation shows is that the distribution of event types on the A interface appears to be the most discriminative.

For each day of data, a separate k-means model with a fixed number of 7 clusters has been built. The centroids of the 7 clusters for each day, reduced to the three features which demonstrated the highest entropy, are shown in Figure 5.9. This figure allows us to observe once more the difference between multiple normal and the anomalous day: Most cluster centroids are positioned around a diagonal line. For the anomalous day, two outlying cluster centroids with a higher distance to the diagonal can be observed. These two cluster centroids were also very prominent outliers when considering different subsets of features, unlike the outliers from the normal day datasets, confirming that in fact the outliers in the anomalous dataset are

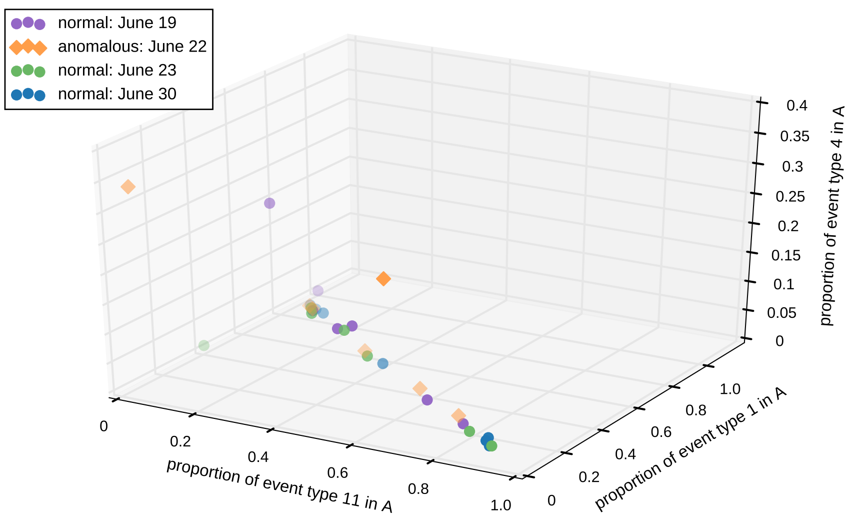


Figure 5.9: Cluster centroids in k-means for models built from the data for normal and anomalous days, reduced to the three most discriminant dimensions for cluster centroid positions.

anomalous not only in the subset of features that is visualized in Figure 5.9. In conclusion, k-means is able to detect known anomalies, but it requires many processing steps to find and analyze the results. Further, k-means requires iterating over the dataset multiple times (here, 20 iterations were used to build the models). This is costly and multiple iterations over the data are especially unsuited for real-time applications.

5.2 Scalability of the Algorithms

We now turn to evaluating the scalability of our anomaly detection system. As our system must be able to scale out and efficiently cope with large amounts of data, we evaluate the processing time of the system given a constant data streaming rate with a varying number of executors (nodes) (Section 5.2.1) and a fixed number of executors with a varying volume of batch input data (Section 5.2.2).

5.2.1 Scale-Out in Streaming

Parallel processing is a key feature of big data infrastructures. In order to evaluate the scalability of our implementation, we conduct an experiment with a varying number of processing executors. Since we are running on top of YARN, each executor corresponds to an isolated memory partition. In our case, we consider memory partitions of 4 GB. All the following experiments were executed during business hours, where we can observe a relatively constant data rate.

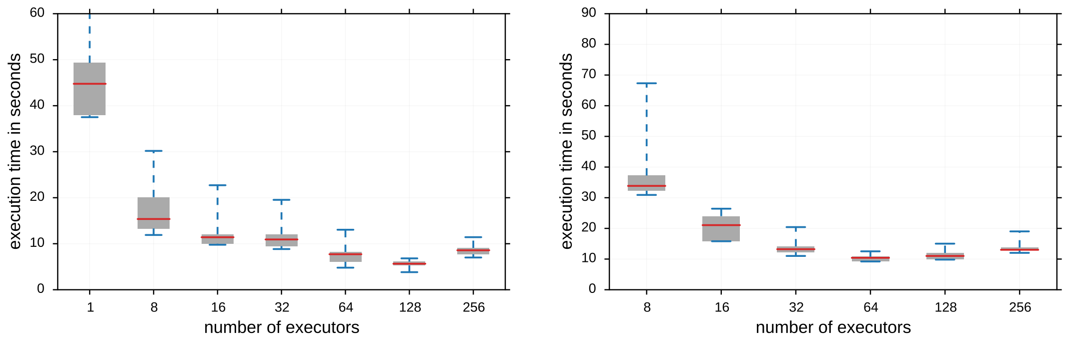
Since the focus of this work is on real-time anomaly detection, this experiment is conducted on the stream implementation of the anomaly detection system described in Chapter 4. Each stream is consumed in parallel by nine receivers, as illustrated in Figures 4.3 and 4.4. These receivers do not run any task other than receiving data. Executors used for receiving data do not process the data any further and immediately distribute the data to other executors. In order to isolate the performance of the processing from the data receiving, we do not consider the stream consumers, but only the executors performing the transformation and the subsequent computations. Thus, we add nine executors for each stream that is being consumed to the total number of executors, but only consider the additional executors for the scale-out experiments.

Since we are interested in the ability of the system to process the data streams in real time, all experiments have been conducted over a period of 20 minutes. Micro-batches, including those created by windowing, should on average be computed within very shorts periods of time, ideally well below the duration in which data are collected (recall Figure 3.3—micro-batches are processed sequentially and we need to prevent data from waiting in the queue and potentially being lost). If the execution of the micro-batch task exceeds a predefined micro-batch time upper bound, then we switch to a different configuration.

Relative Entropy Scalability

In order to evaluate the scalability of the algorithm for computing the relative entropy $D(P_t \| Q_{t-\Delta t})$, we choose a Δt of 60 seconds, i.e., we compute the relative entropy between two adjacent windows of 60 seconds duration.

The experiment in Figure 5.10(a) displays the processing time with an increasing number of executors. This experiment shows that while using only one executor for processing the data, the algorithm terminates on average below the duration of a micro-batch, such that the relative entropy on the stream is computed in real time. The execution time and the variance decrease by adding more executors, but increases after reaching an optimum at 128 executors due to network overhead caused by node management and data shuffling.



(a) Computing relative entropy on the A stream with a micro-batch duration of 60 seconds.

(b) Computing Pearson correlation between the A and IuCS streams with a window duration of 90 seconds.

Figure 5.10: Streaming mode: micro-batch processing times per number of executors.

Pearson Correlation Coefficients

Given the size of the 3G data and the complexity of receiving and processing two streams at the same time, we consider only events from one LAC for the following scalability experiment. The experiment in Figure 5.10(b) displays the micro-batch processing time as a function of the number of executors for computing the Pearson correlation $r(X, Y)$. We limit this experiment to the respective batch duration considered. In the case of Pearson correlation, the processing of the streams using fewer than eight executors was consistently unable to terminate within the given 90 second window duration. Therefore, these configurations are not shown in Figure 5.10(b). The experiment of Figure 5.10(b) shows that the average execution time decreases by adding more executors, until reaching a point where the executor management overhead decreases the performance. For the Pearson correlation algorithm, this point is reached at 64 executors, after which execution speed does not increase any further.

5.2.2 Scalability to Increased Data Quantities

The experiments in this thesis were conducted on 2G and 3G network monitoring events. Newer generations of mobile telecommunication protocols, currently 4G (and in the future 5G), have higher throughput and provide greater monitoring precision, leading to greater amounts of data to be processed. Given that we will be interested in applying our algorithms to these newer technologies as well, we assess the adaptability of the processing performance to larger quantities of data in batch mode.

In order to evaluate the scalability of the algorithms to large volumes of data (measured in hours), we compare different loads. Unlike the previous experiments, where we have considered the relative entropy and Pearson correlation separately, the two measures are computed jointly in the following experiment. The number of executors was fixed to 80 and each executor was configured to 4 GB of memory.

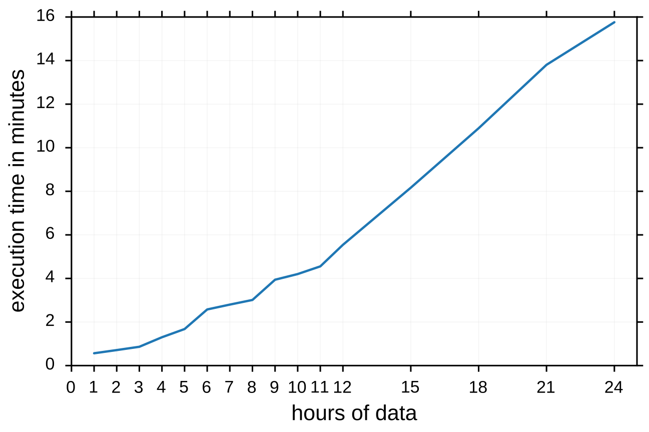


Figure 5.11: Batch mode: joint computation of relative entropy and Pearson correlation: processing time per amount of data in hours.

The computation times for varying data quantities are shown in Figure 5.11. We observe that the processing time scales linearly with the data quantity. Taking around 15 minutes for one day of data, the system is able to process the data and compute the two anomaly detection measures within reasonable time. With limited computational resources, we cannot afford computationally complex processes to handle the data. However, the usage of the shared resources on which anomaly detection is performed varies throughout the day. In cases where periodical batch processing is required, the batches can be processed during idle resource times, for which 15 minutes time windows sound reasonable in our case.

Chapter 6

Discussion

In Chapter 5, we experimentally evaluated the accuracy and scalability of two proposed measures for anomaly detection, relative entropy and Pearson correlation coefficients. In order to assess the accuracy for detecting known anomalies, we used data for known real-world events as well as simulation. We showed that one of our proposed measures—relative entropy—allows us to detect anomalous events related to users' mobility in the form of a high proportion of high values for D, the relative entropy, both between adjacent windows and with fixed time intervals. We found that Pearson correlation is not a suitable measure for detecting anomalies that are of a gradual nature. On the other hand, when simulating an abrupt change in the form of a hardware component's failure, Pearson correlation coefficients show a significant decrease against the highly correlated baseline for normal behavior. Hence, by continuous monitoring over the available data streams, the detection of different types of anomalies is feasible by using both proposed measures simultaneously.

In Table 6.1, we compare our proposed techniques with two state-of-the-art techniques: counting the number of events in an area and clustering feature vectors obtained from windows over the data. In order to evaluate the suitability and compare the different approaches, four dimensions were considered which are relevant for choosing a suitable long-term anomaly detection technique. These dimensions for comparison are the following:

- *Gradual change:* ability to detect changes happening gradually over time, such as human-behavior induced events or gradual failures in the form of increasing event loss or duplication.
- *Abrupt change:* ability to detect abrupt technical changes, caused for example by failing hardware components.
- *Spatial granularity:* anomaly detection accuracy for different levels of geographical granularity—global, regional, and local—with an impact on the efficiency: choosing a

fine granularity, e.g., maintaining a measure over time for each cellsite, implies that a large number of distinct measures has to be computed and stored at any time.

- *Efficiency:* focusing on stream processing, this aspect includes the processing duration, the scalability to parallelization and to larger quantities of data, as well as the efficiency of the algorithms in terms of resource usage, e.g., memory consumption.

Types of anomalies We have evaluated the suitability of our metrics for detecting anomalies of different causes and origins, considering the known types of anomalies that have been enumerated in Section 3.3.3. Relative entropy and both state-of-the-art methods, counting the events and k-means, are suitable for detecting gradual change. Both Pearson correlation and measuring the volume of activity (at a local scale) allow us to detect anomalies as a result of abrupt change, e.g., from infrastructure failures. This results from the quantification of activity in absolute terms for both methods.

We did not evaluate the suitability of relative entropy for detecting abrupt change. Relative entropy is computed between the distribution of the event types, which is based on the relative proportion $P(i)$ (cf. Equation (3.2)) of each event type i at the current time. When losing data from a larger region, the distribution, i.e., the relative proportions of the event types, do not change since the counts per event type experience a uniform downscaling. For this reason, the proportions $P(i)$ will remain the same, such that these types of anomalies—uniform event loss—are not visible from relative entropy or any other measure which relies on relative event proportions for anomaly detection, such as k-means.

Spatial granularity Both proposed techniques, relative entropy and Pearson correlation, are able to detect anomalies on a local, regional, and global scale. For Pearson correlation, the granularity depends on the area that is affected by the failure. Typically, these are larger regions since the physical components deal with aggregates of cells. Thus, computing the measure at a fine granularity, a failure results in cessation of the transmission of any events for this area, resulting in an even lower correlation. However, this fine granularity is not necessary for detecting the failure. Thus, for the sake of efficiency, a regional partitioning or global computation of the correlation is preferable.

The basic counting approach is limited to detecting anomalies locally at the cell level, i.e., even major anomalies are not visible on the regional scale or at nearby cells, as we saw in the case of the railway traffic disruption. This limitation has further implications: semi-automated monitoring, for example through visualizations of the time series data, becomes difficult due to the high number of distinct time series (one for each cellsite). The considered anomalies typically affect more than one cell, requiring us to be able to detect anomalies at an aggregated level; this approach displayed the anomaly strictly at a per-cell granularity.

While k-means clustering did produce observable outliers for anomalous data on a regional scale, this approach was not able to produce distinguishable outliers at a global scale for the same event that we were able to observe on a global scale using relative entropy.

Efficiency We evaluated the computation times for our proposed methods in streaming. In fact, Spark's built-in recovery mechanisms guarantee that all data will be processed, though it

Table 6.1: Comparison of evaluated methods for anomaly detection along relevant dimensions.

Method	Gradual change	Abrupt change	Spatial granularity	Efficiency
Relative entropy	yes, for events linked to human behavior	not evaluated, but not in the case of a uniform down- or upscaling (e.g., hardware or software failures)	both locally and globally	streaming: 5.7 seconds per 60-second micro-batch (median); batch: scales linearly to the amount of data
Pearson correlation	no, not for events linked to human behavior or when simulating a significant gradual duplication of events	yes, for hardware failures	at any granularity since (potentially failing) physical devices cover large regions	streaming: 10.4 seconds per 90-second micro-batch (median); batch: scales linearly to the amount of data
Volume of activity	yes, for human behavior changes, but every anomalous scenario represents itself differently and it is thus hard to learn a model for anomalous sequences	not evaluated, but likely to be visible given that the transmission of events ceases for some cells	only applicable very locally (cellsite level), does not detect any changes on a regional level	not evaluated, but the processing is at least as efficient as our proposed measures due to the low complexity of counting; on the other hand, the need to perform anomaly detection individually for each cell is inefficient due to the high number of distinct metrics
***k*-means**	yes, for human behavior changes	no, not when using relative event proportions as features	regional; no outlying cluster centers could be observed globally	batch: around 2.5 minutes per model (using 7 cluster centers and 20 iterations) per day of data; streaming: not evaluated, but likely inefficient due to the large number of iterations over the data for updating models over time

may require recomputing some partitions and hence imposing some overhead computations. Using the ideal number of executors—128 and 64 respectively—we reach median processing times far below the micro-batch duration (respectively 55 and 80 seconds less), such that fault recovery is feasible. We also obtained a low variance with an increased number of executors, which helps to guarantee an upper bound for the processing time under normal circumstances and less vulnerability in case of failures. Increasing the parallelism beyond the ideal number of executors does not speed up the processing any further in any of the implementations. In our deployment, this is caused by the overhead of the Spark master when managing a large number of executors and distributing data between them through network connections, these becoming the bottlenecks.

The range of suitable configurations, regarding the number of executors, for computing the Pearson correlation between two streams is smaller than for computing relative entropy on one stream. This observation leads to two conclusions. Firstly, we cannot have too few executors since we are simultaneously considering two streams, of which one (3G voice) is notably larger than the other stream (2G voice). The 2G voice stream is the only stream in use for evaluating the relative entropy. Secondly, the optimum performance is reached at fewer executors, due to greater quantities of data being shuffled between executors, generating more overhead at the Spark master than in the relative entropy implementation.

We were also able to demonstrate linear scalability in terms of the amount of data. These results indicate that the algorithms, their implementation, and the Spark processing are efficient, in terms of computing in real-time, computing larger batches, and in scaling horizontally. It appears that a significant fraction of the processing time is caused by data shuffling and executor management.

While the event counting approach has only been implemented over batch data for evaluation using known anomalous events, its implementation—counting the number of distinct events during a period—is simpler than our proposed methods. On the other hand, the need to maintain a large number of distinct values, one for each cell, makes this approach inefficient in terms of resource usage.

It should be noted that k-means is a relatively costly algorithm that requires a large number of iterations over the data. The computation time of k-means does not yet include the feature extraction and transformation steps. Further, this is only for building the model, not for classifying instances. It would need to be investigated whether this approach could be applied in streaming, as in general, multiple iterations are not suited for streaming scenarios. Also, our system computes two measures jointly, providing a more flexible way to detect different anomalies resulting from different events. The insights obtained from k-means may possibly be obtained through less costly methods, such as the ones proposed as part of our anomaly detection system. In addition to the issues with efficiency, it should be further noted that there are some limitations to performing k-means clustering on the monitoring data. The monitoring data are essentially multidimensional time series which are highly correlated. The network traffic logs that were used in the related work proposing this method, on the other hand, have discrete entries.

In summary, by comparing our proposed system against the existing methods, it can be observed that relative entropy detects similar anomalies to k-means—in large parts due to the similar choice of features. It should be pointed out that the choice of features for k-means, at this point, requires prior computation similar to what is performed in our suggested measures. A more naïve approach, e.g., translating each event to a feature vector directly from its set of attributes, would yield a much worse anomaly detection accuracy. Relative entropy further offers an improvement over k-means regarding the efficiency (relative entropy requires only one iteration over the data) and regarding the ability to detect anomalies at the global scale.

Correspondingly, Pearson correlation and the approach using event counts detect similar types of anomalies. It should however be pointed out that, by counting the number of total events without correlating between streams, we can expect to observe false positives, for example as a result of highly local phenomena that do not correspond to real events. On the other hand, Pearson correlation performs at a higher level in the topology, therefore improving in efficiency, and incorporates domain knowledge about the telecommunications protocols (users switch between versions of the protocol and thus appear in different interfaces in the same area under normal circumstances).

Limitations As a result of the lack of ground-truth data, especially anomalous data, the work in this thesis has some limitations and aspects that could not be addressed.

When identifying anomalies, we used thresholds based on the number of measures deviating from the standard deviation of the measure under normal circumstances by a parameter k. These thresholds and parameters vary greatly between the different events and set-ups, meaning that the choice of parameters needs to be determined within the given context. It is at this time difficult to determine the significance of the respective differences when comparing between anomalous data and a baseline. For example, in the scenario in Figure 5.2, the difference on a global scale may not be significant but instead could also occur naturally as a difference between two normal days. In order to draw meaningful conclusions and perform a statistically sound automated anomaly detection, further validation and training data would be necessary.

Another limitation concerns the integration of the current implementation within the overall big data infrastructure. All the experiments in this thesis have been conducted in a controlled setting, running no more than approximately three hours at a time. The stability of the stream processing system for running permanently could hence not be evaluated, though it appeared stable at most times when provided with sufficient resources. Moreover, all output of the system, the measures, were continuously written to logs or files to be analyzed, e.g., by plotting the values, and not to a time series database with automatically updated front-end, as had been indicated in the schema in Figure 4.1.

CHAPTER

7

Conclusion

This thesis presented an anomaly detection system for the purpose of performing DQT over high-velocity streams of telecommunications monitoring data. In the implementation of the system, we leveraged general measures from statistics and information theory and applied them for the purpose of anomaly detection. These measures have been implemented in Spark and Spark Streaming, thus enabling DQT both on data streams and on data at rest. These general measures fulfill the requirement by Swisscom, as they can be, given a suitable choice of features that are used over the streams, adapted to other domains and types of data in the future, e.g., clickstreams. The implementation is flexible and robust in terms of detecting anomalies that occurred on different spatial and temporal scales, since we can consider any subset of the network topology, as well as any duration for computing the measures over subsequences of the stream or the batch data.

The results of the empirical evaluation show that:

- *relative entropy* is suited to detect gradual changes in human behavioral patterns caused by a disruption at one point in time, with the effect gradually increasing and lasting for multiple hours; although using relative proportions of event types, relative entropy does not detect abrupt failures;
- *Pearson correlation* enables the detection of abrupt hardware failures but does not detect any gradual changes; and
- compared to state-of-the-art techniques, the proposed system for anomaly detection is superior in terms of accuracy and efficiency.

Since we require a system that is scalable, we evaluated this desired property both in terms of the amount of data and in terms of the number of executors that are assigned to the processing tasks. Those two aspects are highly relevant because first, the volume on the streams will increase over time with newer generations of technology and more detailed monitoring records, and because secondly, future clusters will consist of even more machines, requiring applications to be able to handle parallel and distributed processing. While both scalability aspects are inherent properties provided by Spark, their availability depends on the correctness of the implementation. We showed that the implementation scales with the number of parallel nodes until reaching an optimum, after which management overhead prevents the performance from being improved. The processing time for increasing volumes of data scales linearly.

Following from these results, it can be stated that Spark and Spark Streaming provide the efficiency and stability to work in production environments, although the configuration of the parameters, such as the number of executors and the spatial/temporal granularity, needs to be carefully evaluated through experiments. Using the proposed measures in Spark and Spark Streaming, customers' need for a measure of the consistency of the data quality can be satisfied through presenting the output of the measures, e.g., for an area of interest, although some further work is required in terms of automated anomaly detection.

Future Work The limitations of our system that were pointed out in Chapter 6 lead to some future work based on the proposed anomaly detection system. From a technical perspective, an integration into the big picture of the big data ecosystem at Swisscom would require a continuous output of the measures to the time series database that is already being used for monitoring. We are also interested in adapting our system to perform anomaly detection over bigger data streams, such as GPEH (3G) and S1-MME (4G), potentially requiring some optimizations of the algorithms in order to deal with the increased volume and velocity, e.g., in the form of randomly sampling the events on the streams, or by replacing the costly sequence of distinct and count operators in the Pearson correlation algorithm with the probabilistic HyperLogLog data structure. Speeding up this bottleneck potentially enables the use of more advanced, more time-consuming correlation methods that are able to detect nonlinear correlation without making the assumption that the data are normally distributed.

The choice of features—event type distributions and distinct anonymized user IDs—is not exhaustive and in future work the anomaly detection can be made more robust by adding anomaly detection over different features. For example, one might imagine that the number of events per user within a timespan is normally distributed but the distribution might shift or change in the case of failures. Given sufficient resources, additional anomaly detection pipelines running in parallel can support the anomaly detection: If anomalies are detected by multiple measures, the likelihood of it being a true anomaly increases.

In order to become a fully automated anomaly detection system, our implementation of the automatic detection of anomalies requires more testing using ground-truth data for determining thresholds and parameters for anomaly alerting. For example, one might consider using resampling techniques to determine the statistical significance of an anomalous measure given the previous information. Another option could be the use of machine learning techniques, such as classification rules, that are learned from annotated ground-truth data at different granularities both spatially and temporally. As a side-effect, this would allow the system to automatically output the type of anomaly along with the alert.

Acknowledgements

First of all, I want to thank my supervisors: Philippe Cudré-Mauroux for approaching me with this project and for his valuable advice and feedback. Michal Piorkowski for initiating the collaboration with Swisscom, thus giving me the opportunity to work with real-world big data, and for guidance throughout my project. Mourad Khayati for the close collaboration and for teaching me valuable academic skills.

Thanks to my colleagues at Swisscom for making me part of a team working on exiting new topics, for sharing their knowledge and experiences, and for the enjoyable lunch and coffee breaks. In particular, I would like to thank Marc Zimmermann and Thibaud Chardonnens for practical help with the software and infrastructure and for giving me plenty of food for thought.

On a personal note, my gratitude goes to Simon for being there during my ups and downs and his dedication to proofreading. My special thanks are extended to my family for their continued support and encouragement.

Bibliography

[1] B. Babcock, M. Datar, and R. Motwani, “Load shedding techniques for data stream systems,” in *Proceedings of the 2003 Workshop on Management and Processing of Data Streams (MPDS 03)*, vol. 577, 2003.

[2] F. Buytendijk, *Hype cycle for big data, 2014*, 2014. [Online]. Available: `https://www.gartner.com/doc/2814517` (visited on Jul. 15, 2015).

[3] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, “Improving data quality: consistency and accuracy,” *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)*, pp. 315–326, 2007.

[4] G. Cormode and S. Muthukrishnan, “What’s new: finding significant differences in network data streams,” *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*, vol. 3, pp. 1534–1545, 2004.

[5] G. Cormode and S. Muthukrishnan, “An improved data stream summary: the count-min sketch and its applications,” *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[6] T. Dasu, S. Krishnan, S. Venkatasubramanian, and K. Yi, “An information-theoretic approach to detecting changes in multi-dimensional data streams,” in *Proceedings of the 38th Symposium on the Interface of Statistics, Computing Science, and Applications (Interface ’06)*, 2006.

[7] M. Datar, A. Gionis, P. Indyk, and R. Motwani, “Maintaining stream statistics over sliding windows,” *SIAM Journal on Computing (SICOMP)*, vol. 31, pp. 1794–1813, 2002.

[8] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” in *Sixth Symposium on Operating System Design and Implementation (OSDI ’04)*, San Francisco, CA, 2004.

[9] E. W. Dereszynski and T. G. Dietterich, “Probabilistic models for anomaly detection in remote sensor data streams,” in *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI 2007)*, 2007, pp. 75–82.

[10] X. L. Dong, E. Gabrilovich, K. Murphy, V. Dang, W. Horn, C. Lugaresi, S. Sun, and W. Zhang, “Knowledge-based trust: estimating the trustworthiness of web sources,” *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 938–949, 2015.

[11] E. Dumbill, *What is big data? an introduction to the big data landscape.* 2012. [Online]. Available: `https://beta.oreilly.com/ideas/what-is-big-data` (visited on Jul. 20, 2015).

[12] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *2007 Conference on Analysis of Algorithms (AofA '07)*, 2007, pp. 127–146.

[13] S. Geisler, S. Weber, and C. Quix, "Ontology-based data quality framework for data stream applications," in *Proceedings of the 16th International Conference on Information Quality (ICIQ 11)*, 2011, pp. 145–159.

[14] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han, "Outlier detection for temporal data: a survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 9, pp. 2250–2267, 2014.

[15] D. J. Hill, B. S. Minsker, and E. Amir, "Real-time bayesian anomaly detection for environmental sensor data," *Water Resources Research*, vol. 45, no. 4, 2009.

[16] Y.-T. Huang, F.-T. Cheng, and Y.-T. Chen, "Importance of data quality in virtual metrology," in *32nd Annual Conference of IEEE Industrial Electronics (IECON 2006)*, 2006, pp. 3727–3732.

[17] M. G. Kahn, M. A. Raebel, J. M. Glanz, K. Riedlinger, and J. F. Steiner, "A pragmatic framework for single-site and multisite data quality assessment in electronic health record-based clinical research," *Medical Care*, vol. 50, pp. 21–29, 2012.

[18] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer, "Profiler: integrated statistical analysis and visualization for data quality assessment," *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces (AVI '12)*, pp. 547–554, 2012.

[19] A. Klein, H. H. Do, G. Hackenbroich, M. Karnstedt, and W. Lehner, "Representing data quality for streaming and static data," in *Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007)*, 2007, pp. 3–10.

[20] F. Korn, S. Muthukrishnan, and Y. Zhu, "Checks and balances: monitoring data quality problems in network traffic databases," in *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, 2003, pp. 536–547.

[21] S. Kullback and R. A. Leibler, "On information and sufficiency," *The Annals of Mathematical Statistics*, vol. 22, pp. 79–86, 1951.

[22] D. Laney, "3D data management: controlling data volume, velocity, and variety," Tech. Rep., 2001. [Online]. Available: `http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf` (visited on Jul. 14, 2015).

[23] W. Lee and D. Xiang, "Information-theoretic measures for anomaly detection," in *2001 IEEE Symposium on Security and Privacy (S&P 2001)*, 2001, pp. 130–143.

[24] X. Li and J. Han, "Mining approximate top-k subspace anomalies in multi-dimensional time-series data," in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)*, 2007, pp. 447–458.

[25] Q. Ma, S. Muthukrishnan, and M. Sandler, “Frugal streaming for estimating quantiles,” in *Space-Efficient Data Structures, Streams, and Algorithms*, vol. 8066, Springer Berlin Heidelberg, 2013, pp. 77–96.

[26] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, “Semantics of data streams and operators,” in *Proceedings of the 10th International Conference on Database Theory (ICDT 2005)*, Edinburgh, UK, 2005, pp. 37–52.

[27] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. Greenwich, CT: Manning Publications Co., 2013.

[28] J.-N. Mazón, J. J. Zubcoff, I. Garrigós, R. Espinosa, and R. Rodríguez, “Open business intelligence: on the importance of data quality awareness in user-friendly data mining,” in *Proceedings of the 2012 Joint EDBT/ICDT Workshops (EDBT-ICDT ’12)*, 2012, pp. 144–147.

[29] G. Münz, S. Li, and G. Carle, “Traffic anomaly detection using k-means clustering,” in *GI/ITG Workshop MMBnet*, 2007.

[30] S. Muthukrishnan, “Data streams: algorithms and applications,” Rutgers University; AT&T Labs–Research, Tech. Rep., 2005, pp. 1–39.

[31] J. Naisbitt, *Megatrends: Ten New Directions Transforming Our Lives*. New York: Grand Central Publishing, 1988.

[32] J. E. Olson, *Data Quality: The Accuracy Dimension*, M. Kaufmann, Ed. Elsevier Science, 2003.

[33] O. Papapetrou, M. Garofalakis, and A. Deligiannakis, “Sketch-based querying of distributed sliding-window data streams,” in *Proceedings of the VLDB Endowment*, vol. 5, 2012, pp. 992–1003.

[34] L. Rettig, M. Khayati, P. Cudre-Mauroux, and M. Piorkowski, “Online anomaly detection over big data streams,” submitted for publication.

[35] S. Ryza, *Apache spark resource management and YARN app models*, 2014. [Online]. Available: `http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/` (visited on Jul. 20, 2015).

[36] B. Saha and D. Srivastava, “Data quality: the other face of big data,” in *Proceedings of the 30th International Conference on Data Engineering (ICDE 2014)*, Chicago, IL, 2014, pp. 1294–1297.

[37] P. Saporito, *2 more big data V’s – value and veracity*, 2014. [Online]. Available: `http://www.digitalistmag.com/big-data/2-more-big-data-vs-value-and-veracity-01242817` (visited on Jul. 14, 2015).

[38] The Apache Software Foundation, *Spark streaming programming guide*. [Online]. Available: `http://spark.apache.org/docs/1.3.1/streaming-programming-guide.html` (visited on Jun. 3, 2015).

[39] R. L. Thorndike, “Who belongs in the family?” *Psychometrika*, vol. 18, no. 4, pp. 267–276, 1953.

[40] Q. Wu and Z. Shao, “Network anomaly detection using time series analysis,” in *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services (ICAS-ICNS 2005)*, 2005, p. 42.

[41] W. C. Young, J. E. Blumenstock, E. B. Fox, and T. H. Mccormick, “Detecting and classifying anomalous behavior in spatiotemporal network data,” in *The 20th ACM Conference on Knowledge Discovery and Mining (KDD ’14), Workshop on Data Science for Social Good*, New York, NY, 2014.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI ’12)*, 2012, p. 2.

[43] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: cluster computing with working sets,” in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud ’10)*, 2010, p. 10.

[44] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, “Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters,” in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing (HotCloud ’12)*, 2012, p. 10.

[45] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: fault-tolerant streaming computation at scale,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP ’13)*, 2013, pp. 423–438.

[46] J. Zhang, M. Lou, T. W. Ling, and H. Wang, “HOS-miner: a system for detecting outlying subspaces in high-dimensional data,” in *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004)*, Toronto, Canada, 2004, pp. 1265–1268.