CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

Streaming Novelty Detection in Telemetry Data

Holger Nießner

Thesis presented for the degree of

Space Master - Joint European Master in Space Science and Technology

Supervisor CTU: Prof. Tomas Pajdla

Supervisor Solenix: Yann Voumard

Czech Technical University, Prague, Czech Republic

Luleå University of Technology, Kiruna, Sweden

January 2017

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Holger Niessner**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Diploma Thesis: **Novelty Detection in Spacecraft Telemetry Data**

Guidelines:

Spacecrafts capture, store and transmit vast amounts of data, that can not be processed by human operators anymore. To make sense of the data, design a process for discovering patterns and oddities in off-line data. Given monthly or more frequent data inputs, the program should learn how to classify it and give reports on new findings. This thesis is done in cooperation with Solenix [1]

1. Get familiar with Apache Spark and Elasticsearch
2. Learn the features of relevant Satellite mission and compile abstract model
3. Implement single machine learning algorithm on one-off data
4. Implement a pipeline to process data on a monthly basis

Bibliography/Sources:

[1] Kelleher, Mac Namee, D'Arcy: Machine Learning for Predictive Data Analysis
[2] Dude, Hart, Stork: Pattern Classification
[3] Karau, Konwinski, Wendell: Learning Spark
[4] Abbott: Applied Predictive Analytics: Principles and Techniques for the Professional Data Analyst

Diploma Thesis Supervisor: Ing. Tomáš Pajdla, Ph.D.

Valid until the summer semester 2016/2017

prof. Ing. Michael Šebek, DrSc.
Head of Department

L.S.

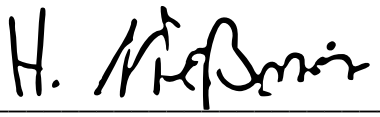
prof. Ing. Pavel Ripka, CSc.
Dean

Prague, March 11, 2016

I, Holger Nießner confirm that the work presented in this thesis titled 'Streaming Novelty Detection in Telemetry Data' is my own.

I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualifcation at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed 

Date 25.05.2017

Abstract

Modern satellites produce a large amount of telemetry data (> 10.000 parameters) among which the most important ones are monitored by operators during passes. Processing this amount of data in real time exceeds the capability of human-based analyses. This has led to a rise in the so-called big data and machine learning systems that learn from this data.

In the last years, solutions have been developed for detecting previously unknown situations in these parameters and in the system state in general. With the desire to analyse data in real time an approach that leverages open source technologies and distributed computing is desired.

This thesis gives a solution to the problem of live outlier detection in the environment of Space missions, especially the processing in the time-constraint of a single overpass.

Table of Contents

List of figures

List of tables

Chapter 1

Introduction

> The grandest discoveries of science have been but the rewards of accurate measurement and patient long-continued labour in the minute sifting of numerical results. - Lord Kelvin

This chapter gives an introduction to the thesis including main motivation, questions and objectives.

1.1 Problem Statement

Modern satellites produce large amounts of telemetry data. The most important parameters are monitored by operators during connections with a ground-station, while others are analysed by engineers later-on if needed.

Every year satellites increase in complexity and gain more computing power. This leads to an ever increasing amount of data that needs to be made available, analysed and understood. Novel situations, including onboard failures

or foreign circumstances, need to be detected. These events are hard to detect for human operators, because they can be spread across a huge number of parameters in different time scales. Machines that watch the state and health of a satellite can hardly be trained to recognise situations that are novel when they happen.

Technical solutions have been developed for detecting previously unknown situations. The existing approach regarding the data processing is to send the data from the satellite and wait until the result is computed before the next overpass occurs. A more timely reaction is desired, preferably while contact to the satellite remains. Therefore a system has to be devised, which can process the data as a stream and give immediate feedback on detected novelties.

Novelty detection algorithms have been researched with live processing in mind. Previous approaches in offline processing had a straightforward implementation, because all data was available at the time of analysis. In streaming algorithms the whole data set can not be reprocessed every time new information arrives. New techniques rely on dimensionality reduction to store the knowledge about previous states in a highly reduced manner. This allows faster computing with almost no loss in accuracy.

Big data analysis tasks often require more processing power than a single computer can provide. A solution is to combine servers into large computing pools. They provide easy access for a developer, who needs no knowledge of the underlying structure. Opposed to the concentrated nature of a single server, this pool is a distributed computing environment. Algorithms which are executed in it, have to be adapted to its nature in order to leverage the increased computing power.

From the initial problem scenario and the given environment, the objectives for this thesis can be summarised as: Design a system which is able to detect outliers in streaming telemetry data, especially in the time-constraint of a single overpass.

1.2 Thesis outline

The following thesis is organised as follows: *Chapter 2* gives a summary of the necessary theoretical background with *Chapter 3* explaining the given test data. Requirements for the thesis project are given in *Chapter 4*. The implementation of the algorithms and the technology stack is described in *Chapter 5*. *Chapter 6* then reveals the benchmark and its results. It is followed by the conclusion in *Chapter 7* and an outlook on future work in *Chapter 8*.

Chapter 2

Theory: Concepts & Definitions

This chapter gives a brief overview of the theoretical foundation for the remaining chapters. It provides necessary definitions of key terms and techniques.

2.1 Timeseries

A collection of organised data over time is called a time series. Most measurements in science and engineering are performed over time. A time series therefore represents a collection of sequential values. Some familiar occurrences are daily stock market prices, monthly unemployment figures or yearly population growth. [1]

In order to gain insights into the process creating these time series, one applies different techniques like outlier detection, prediction, clustering or classification. [2] Applications of these analyses include a wide range of problems such

as intrusion detection, economic forecasting or medical surveillance.

Telemetry or housekeeping data in a satellite is periodically saved. Therefore each measurement, provided by a sensor or derived otherwise, belongs to a time series. This allows us to process telemetry data with existing techniques, such as novelty detection.

The mathematical description of a time series is:

(1) $T = (t_1, t_2, \ldots t_n), t_i \in \mathbb{R}$

For simplicity we will focus only on sequences of scalars, although the techniques are generally just as useful for vector series. Theoretically *t* varies continuously in time, such as a temperature. In practice *t* is sampled to a discrete sequence of data points, which are equally spaced in time following a given *sampling rate*.

A time series is often the result of an underlying process, for example the varying influence of the sun on a temperature. A time series can be *univariate* as in the previous definition or *multivariate* when several series span multiple dimensions in the same time range. They can cover the full range of data and be of considerable length. When talking about *streaming* time series, we define them as semi-infinite series. Every time instant continuously extends the series, like an ongoing temperature measurement saving data every minute.

In order to gain insights into the process creating a time series, one applies different tasks to it, for example outlier detection, prediction, or summarisation [2].

Applications of these analyses include problems such as intrusion detection,

economic forecasting or medical surveillance. In the following sections I will introduce some of these tasks and give a detailed explanation of novelty detection.

2.2 Summarisation

The task of *summarisation* (or segmentation) on time series creates an approximation. It reduces the dimensionality while maintaining essential features. The definition therefore is:

Given a time series $T = (t_1, t_2, \ldots t_n)$, construct a model T' of reduced dimensionality $d' \ll n$ such that T' closely approximates T. $|R(T') - T| < \epsilon_r$, with $R(T')$ as the reconstruction function and ϵ_r as error threshold.

This task aims to minimise the reconstruction error between a reduced representation and the original time series. Several approaches have been researched, the most common one is Piecewise Linear Approximation (PLA) [3], which splits the series into segments and then fits polynomial models to each of them.

2.3 Prediction

The task of prediction tries to create a model to forecast the next few values of a series. The definition is:

Given a time series $T = (t_1, t_2, \ldots t_n)$, predict the *k* next values $t_{n+1}, \ldots, t_{n+k}$ that are most likely to occur.

Prediction is one of the most common tasks in real-life applications, such as stock market forecasting, which relies heavily on a wealth of historic data.

2.4 Novelty Detection

A novelty can be defined as an 'unexpected value or a sequence of values' [4]. One of the first original definitions of an outlier is by [5]: "An observation (or subset of observations) which appears to be inconsistent with the remainder of that set of data." Novelty detection therefore is the ability of a system to recognise a sequence, which appears for the first time or does not fit into the current context. It is an important issue in image scene analysis and intrusion detection. [6] Recognising frauds in the monetary system or faults in mechanical machines are also important applications.

The terms *novelty* and *outlier* will be used interchangeably in this section, as most literature in the field chooses to do so.

The satellite as an enclosed system packed with sensors in a harsh environment provides an interesting case for detecting novelties. Repeating orbits give a periodic baseline against which unexpected behaviour stands out.

A basic visual example is Figure 2.1, which is adapted from [7]. It shows the main distribution in the center and three outliers in the lower right corner.

Novelty detection is a very challenging task, therefore there are different approaches that perform well on different types of data. There is no single best model and success depends on statistical properties of the data and the applied method. [8]

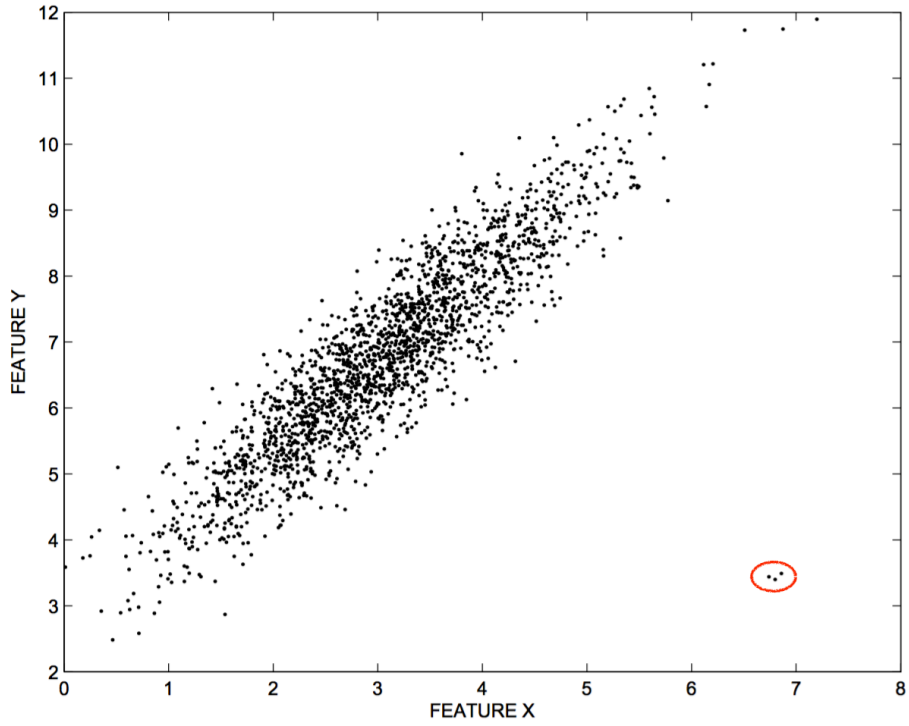


Figure 2.1: Example of a 2D distribution with outliers

Novelty detection is an important task in many safety critical environments as an outlier indicates abnormal running conditions, like an aircraft engine defect or a flow problem in a pipeline. Outlier detection can reveal a fault on a factory production line by monitoring specific features of the products and comparing the real-time data with either the features of normal products or those for faults. Applications like credit card usage monitoring or mobile phone monitoring are needed to detect a sudden change in the usage pattern which may indicate fraudulent usage such as a stolen card or phone.[9]

A straightforward method for novelty detection in time series is based on forecasting. [10] A model is built from the historical values and is used later to predict future values. If a predicted value differs from the observed value beyond a certain threshold, a novelty is detected. The threshold to be used for

detecting novelties is the main problem of detection based on forecasting.

The location of cash withdrawals with a credit-card as a time series, can be used as an example for a threshold problem. Is the threshold range too large, too little outliers are recognised. A malicious withdrawal on another continent would not trigger an anomaly. Is the threshold range too small, e.g. only concerning the card-owners home town, a trip to another city would trigger an anomaly.

Other approaches relying on a decision boundary include both Frequentist and Bayesian approaches, information theory, extreme value statistics, support vector methods, other kernel methods, and neural networks.[11] These methods need a training set that is selected to contain no or very few of the novel class. The output is then compared to a novelty threshold, which usually has to be defined before.

This difficulty of setting a proper threshold has motivated methods like classification, pattern-based and others. These analyse a sequence of a given length and indicate whether they correspond to normal or anomalous behaviour. Most time series are high dimensional, which makes detecting anomaly patterns computationally expensive.

Pimentel et al. write in [11] "The complexity of modern high-integrity systems is such that only a limited understanding of the relationships between the various system components can be obtained. An inevitable consequence of this is the existence of a large number of possible "abnormal" modes, some of which may not be known a priori, which makes conventional multi-class classification schemes unsuitable for these applications. A solution to this problem is offered by novelty detection, in which a description of normality is learnt

by constructing a model with numerous examples representing positive instances (i.e., data indicative of normal system behaviour). Previously unseen patterns are then tested by comparing them with the model of normality, often resulting in some form of novelty score."

The following sections describe techniques to adapt and shape data in beneficial ways. They improve the performance of analysis and machine learning algorithms.

2.5 Normalisation

Normalisation or *scaling* is a common step in the pre-processing of data. It brings different features to one scale, meaning a range of values, variance or length. Not normalising the data often results in worse performance when applying machine learning or data analytics algorithms. [12]

The two most common techniques are Min-Max Normalisation and Z-Score Standardisation. Others are Decimal scaling, Bi-Normal Separation (BNS) feature scaling [13] and Rank normalisation. [14].

2.6 Moving average

We calculate the moving average of the recent time series window, in order to cancel out noise and minor fluctuations. The first approach is to simply compute the average of the last n data points.

(2) $MA(t) = \frac{1}{n}\sum_{i=0}^{n} x_{t-i}$

An incremental formular, with no need to store the past *n* values is given by:

(3) $MA(t+1) = MA(t) - \frac{x_{t-(n-1)}}{n} + \frac{x_{t+1}}{n}$

A more sophisticated technique is the weighted moving average, which multiplies the past values with a growing factor. The current value is taken as is, while the oldest value is decreased the most.

(4) $WMA(t) = \frac{\sum_{i=0}^{n-1} c_i x_{t-i}}{\sum_{i=0}^{n-1} c_i}$

2.7 Dimension reduction

Dimension reduction is the process of reducing the number of variables of a given data set, by obtaining uncorrelated principal variables. It is often used in the fields of machine learning and image processing, to decrease the size of input data for algorithms. The number of principal variables, which store the major characteristics, is usually much smaller than the original size, in the worst-case scenario they are equal. Algorithms can process the result faster and visualisation can be created more easily in 2D or 3D dimensions.

Dimension reduction consists of feature selection and extraction, whereas feature selection is the lesser point in this discussion. We will not limit the analysis to a subset of the data, therefore we focus on feature extraction.

The most common tactic to extract characteristic features is *Principal Component Analysis* (PCA), a way to identify patterns and express the data in order

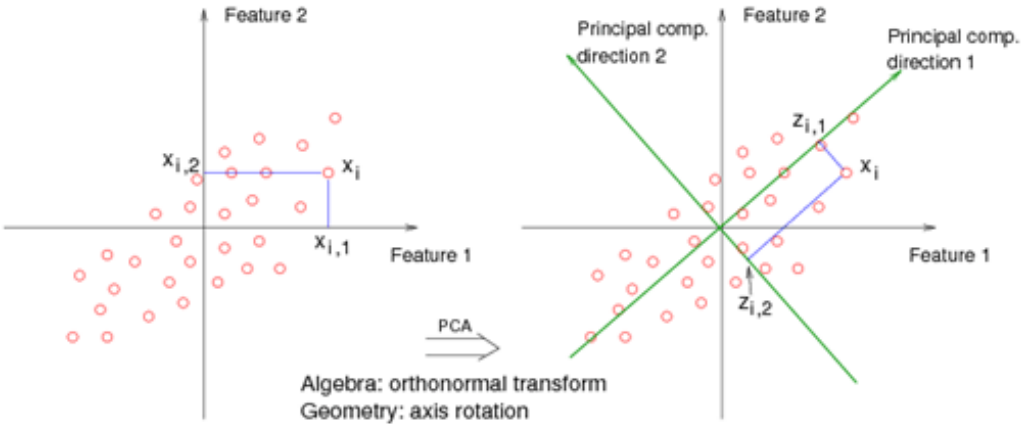


Figure 2.2: PCA transformation on a 2D data set

to highlight their similarities and differences. It captures the linear correlation between variables (or features) and transforms them into fewer linearly uncorrelated hidden variables. The goal is to find the subspace on which the data can be represented the best. There are other approaches which are not limited to the linear space, Manifold learning algorithms such as *Self-organizing Map* (SOM) or *Principal curves*. [15]

2.7.1 Principal Component Analysis (PCA)

PCA is one the most common techniques used in feature extraction. It is used to eliminate insignificant components and reduce the representation of data to the significant elements.[16]

PCA is defined as an orthogonal linear transformation from input data to a new coordinate system such that the greatest variance comes to lie on the first principal component, the second greatest variance on the second component, etc. [17]. Figure 2.2, based on [18], shows the processing of a 2D data set with PCA.

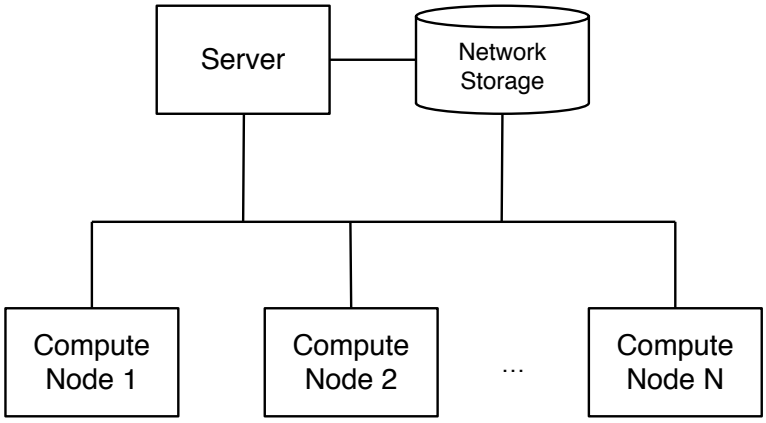


Figure 2.3: Distributed computing architecture

A four-step approach to implementing PCA is:

- Obtain the Eigenvectors and Eigenvalues from the covariance matrix or correlation matrix, or perform Singular Vector Decomposition.
- Sort eigenvalues in descending order and choose the *k* eigenvectors that correspond to the *k* largest eigenvalues where *k* is the number of dimensions of the new feature subspace ($k \leq d$)
- Construct the projection matrix *W* from the selected *k* eigenvectors
- Transform the original dataset *X* via *W* to obtain a *k*-dimensional feature subspace *Y*

Algorithms based on PCA are effective when all data is stored in one location. Improvements have been developed to attenuate this data constraint. With a slight decrease in accuracy

2.8 Distributed computing

A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal. [19] Three significant characteristics of a distributed systems are:

1. concurrency of components Separate servers can work on tasks at the same time. A goal can be achieved faster, if it can be separated into workloads that do not depend on each other.

2. lack of a global clock Each component in a distributed system needs to manage its own clock. Separate clocks will drift apart and timing issues can get introduced. This effect needs to be weakened by keeping the same time in all system with the Network Time Protocol (NTP). [20]

3. independent failure of components A distributed system needs to be fault tolerance to failures of single components or whole subparts of it. If a server fails, the workload needs to be handled by another system and not be dropped completely.

A distributed computing system can have one of many architectures, such as client-server, n-tier or peer-to-peer. Sharing of resources is the main motivation for constructing a system to be distributed. The challenges stem from the characteristics seen above, mainly that algorithms need to leverage the concurrency properly and components can be added, replaced or turned off during normal operation.

A major benefit is the inherent scalability of a distributed system. If set up properly, the system can be scaled to work faster by adding more computation power. This works up to a certain point, as with an increasing number of servers the necessary network traffic to handle communication increases as well.

The following table, which is based on [21], gives a concise summary of distributed computing.

Table 2.1: Distributed Computing Overview

Category	Description
Resource constraints	Memory, computational capacity and communication bandwidth
High communication cost	Communication cost is orders of magnitude more than computation costs
Distributed streaming data	Processing data coming at different rates from multiple distributed sensors
Dynamic nature	Dynamic network topology, frequent communication failures, mobility and heterogeneity of nodes
Large-scale deployment	Scalability issues with traditional outlier detection algorithms
Identifying outlier sources	Make distinction between errors, events and malicious attacks

Chapter 3

Data Review

This chapter gives insight into the data set, provided by the collaboration of Solenix [22] and ESA [23]. The data was collected over a three month period from June until August 2015. Each time series is a sequence of values recorded onboard a satellite in a currently active ESA mission.

Overall there are 4334 time series with 3.798.093.977 data points.

The database consists of five tables, that differ on the detail of value they can store.

Table 3.1: Name, Datatype and Number of entries for tables in database

Name	Datatype	Nr. of entries
floatparamvalues	decimal point	2.865.066.347
ssmallintparamvalues	int	21.410.819
uintparamvalues	unsigned int	103.804.948
umediumintparamvalues	int	74.241.792
usmallintparamvalues	int	733.570.071

Name	Datatype	Nr. of entries
Total		3.798.093.977

Each table consists of rows with *PID*, *datetime* and *value*.

3.1 Values

3.1.1 Range

Values in the data set vary widely. The range in one time series vary between 1092 and 16777215 or between 0 and 1.

This is caused by different units of measurement for each series. Every sensor receives a temperature, voltage, acceleration or else. The enormous difference in scale of time series could effect the performance of algorithms in data mining. Thus, it is one of the problems that we have to deal with in the implementation, with techniques such as normalisation, see Section 2.5.

3.1.2 Sample Rate

All time series are recorded on one satellite. They still have different sampling rates and are not synchronised. The total recording time of data is from 30. June 2015 00:13:20 2015 UTC to 31. August 2015 23:59:58 2015 UTC. During this period, different sensors saved values between 515 and 1482128 times, with a mean of 622055 times.

Chapter 4

Requirements

This chapter introduces the requirements and the reasoning behind them. They were developed in cooperation with Solenix [22], as the industry partner providing insights into the desired product.

The main request is to design a system which is able to detect outliers in streaming telemetry data, especially in the time-constraint of a single overpass.

This gives three requirements. First, the output of the system should be outliers which are novel or differ from normal operation. This inclusive definition of anomalies is desired as false positives can be easily discarded later by a human operator. When forcing the systems to give no false alarms, it would also recognise a smaller number of real outliers. A missed outlier can lead to tremendous damage in the space segment. It is more expensive to miss an outlier than to force the system to give no false alarm. Therefore the designed system should err on the side of false positives.

Second, the operation is streaming or live. New data can come in anytime and

should be processed immediately. The system will be used in Space operations, therefore we will set a requirement based on this environment. A satellite pass is around 30 minutes, with half of it for processing and the other half for human interaction and responding to results of the analysis. If the system gives an outlier signal, there should be enough time for operators to analyse and react to the new information.

The third point is, as described in Chapter 3, the data is in form of time series. No visual or textual information needs to be processed. The analysis is focused on numerical values attached to specific points in time.

The designed system should be able to deliver novelty analysis for several ESA missions, where each one has a different space segment. The approach of one server per mission is not feasible, as it would waste computation capacity, gives a single point of failure and can not adapt to the increasing demand.

Therefore it is desired but not mandatory, to leverage the power of distributed computing (detailed description in Chapter 2.8). The nature of distributed computing has benefits and drawbacks. With concurrency and independent failure of components, we gain faster computing times with more computing power, but need to handle losses of servers as well. This is a major point in evaluating suitable algorithms. Is an algorithm scalable and does it leverage the power of the distributed computing environment as much as possible?

After the analysis is done and the current processing step has stopped, the system needs to give its results. It is not enough to have a single blinking light to signal novelties. An operator must be able to quickly read the result of the analysis, verify them with her domain knowledge and resolve the particular situation. Therefore the output should include the time and duration of an

Figure 4.1: Simple depiction of data pipeline

outlier, as well as the anomalous time series.

Table 4.1 gives the requirements in a structured form:

Table 4.1: Requirements

Category	Description
Input	Telemetry Time series data
Output	Novelties
Mode of Operation	Streaming/Live
Algorithm	Leverage distributed computing
Product	Use system in real application

4.1 Data Pipeline

A data pipeline describes the information flow from an input through transformations to the output. In this section I will outline the desired configuration in a production system, as well as the development setup.

A basic data pipeline can be seen in Figure 4.1. It describes the concept in a general way because several complexities not shown. The distributed nature of the transformations (see Chapter 2.8), as well as different data types for input

and output are not visible.

To consume live streaming data during the development of any analysis system is not feasible. Nothing is gained by accessing current satellite data while creating the system. Nevertheless, the project needs to be tested with real data (see Chapter 3) to assure accurate performance when used in production.

The data pipeline should be simplified first, to build the core algorithms and data handling, and then plug the system into a production environment. This gives less configuration overhead and the developer can concentrate on important questions [24], such as:

1. Is the transformation reproducible?
2. Is the pipeline consistent?
3. Can all the pieces be shaped into a production system?

Regarding 1. and 2.: random seed generators are seeded with fixed values which eliminates algorithmic randomness. The initial test dataset is fixed as well, therefore the results are reproducible and the pipeline is consistent.

Question 3, is it possible to move this scientific thesis project into a production-ready system, is addressed in the next section.

4.2 Development to Production

This section is concerned with the move of the project from a development setup to a production-ready system.

A common pattern in software engineering regarding testing and deployment

is *DTAP*, short for development, testing, acceptance, and production. [25] These describe environments or a cycle of stages that a project goes through.

As tests are an integral part of modern development and the acceptance stage is not relevant for this project, this section discusses development and production.

Development is the environment of an individual developer's workstation. It differs from production in several ways - the target may be a different device, such as a smartphone, embedded system, or headless server. This stage includes development tools like a compiler, integrated development environment, and support software, which will not be part of the final system. With *test-driven development*, short *TDD*, the additional testing stage merges into this phase.

Production is the final environment and it concludes the initial project cycle. The stage is only filled with the necessary code, binaries and configurations, with as little overhead as possible. Deploying to production is the most sensitive step in the entire engineering process.

The next chapter will explain the implementation of these requirements in this project setup.

Chapter 5

Implementation

This chapter will give detailed information about the implementation, building on last chapter's explanation of the requirements and the data pipeline. This will give the reader a high-level overview of the design, as well as in-depth knowledge of configuration and structure.

5.1 Data pipeline implementation

Following the general explanation in chapter 4.1, this section gives information about the specific implementation of the data pipeline.

Figure 5.1 shows details of the transformation in the case of outlier analysis with the Spirit algorithm (see 5.3.1). It shows three steps, from pre-processing through the core algorithm to the outlier analysis. The development and production environments differ in source and destination of the data.

Figures 5.2 and 5.3 depict the environments during development and produc-

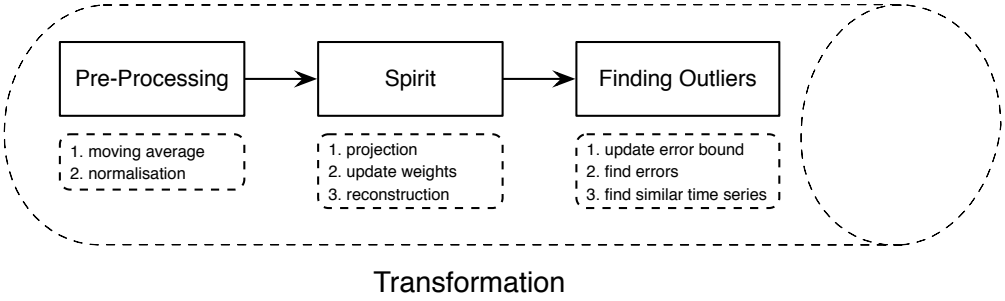


Figure 5.1: Detailed figure of data transformation

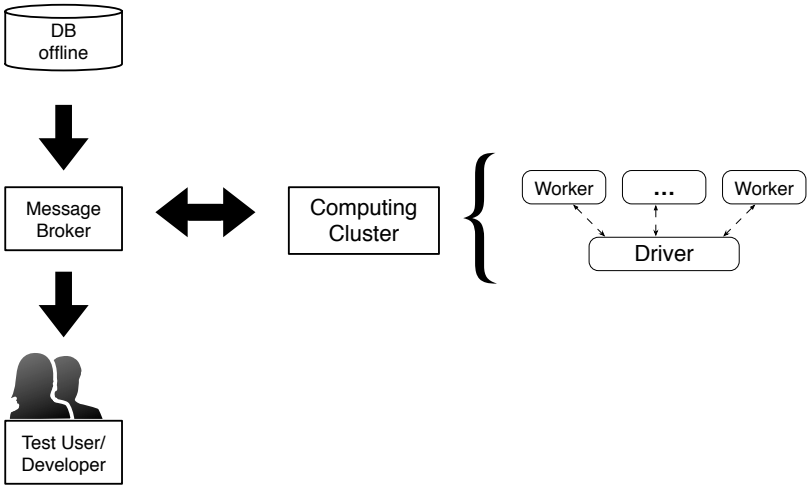


Figure 5.2: Pipeline during development

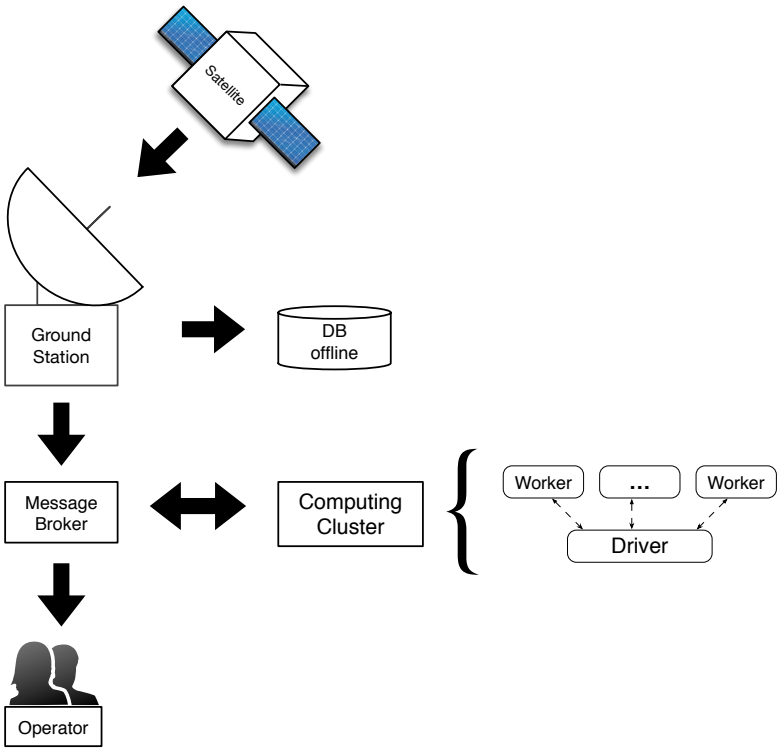


Figure 5.3: Pipeline for production system

tion. One can recognise the differences and similarities at a glance.

The design process was started with the initial requirements in mind (see Chapter 4), to transform the system into a real-world application. This is why the similarities include the messaging broker and computing cluster at the core of both environments. Changing from development to production, the data in- and outputs need to be changed. This is configured at only one single point, the message broker. The computing cluster (see 2.8) can easily adapt to new hardware, more servers and other configurations. The interconnection with the message broker stays the same and during a move to production of no concern.

As small packet format in JSON, the JavaScript Object Notation, was created to facilitate easy compression and decompression across the whole technology stack as described in 5.2. The format is small and simple in order to handle packets in a fast and efficient way. An input data packet is described like this:

```
{
    "ts": "NUMBER",
    "data": [
      {"PID": "NUMBER","value": "NUMBER"},
      {"PID": "NUMBER","value": "NUMBER"},
      {"PID": "NUMBER","value": "NUMBER"}
    ]
}
```

ts gives the timestamp of the following values, which are ordered by their *PID*, the index of each time series.

An output data packet describing an outlier event is described like this:

```
{
    "TS": [
      {"PID": "NUMBER"},
      {"PID": "NUMBER"},
      {"PID": "NUMBER"}
    ],
    "from": "DATE",
    "to": "DATE"
}
```

Following the requirements in Chapter 4, the packet contains:
TS is a list of indices of the time series that contribute to this outlier. *from* and *to* are the dates between which the outlier is active.

5.2 Technology Stack

This section will give additional details on which technologies have been used to implement the outlier detection system. The big data processing engine Spark (section 5.2.1), the programming language Scala and its runtime environment JVM (section 5.2.2) and the message broker Kafka(section 5.2.3) will be covered. DB2Kafka, the custom program which streams data from the database into the message broker, as well as the distributed configuration service ZooKeeper are explained. PlotMeisterTS, the custom script, which visualised outlier data, is introduced with a short example.

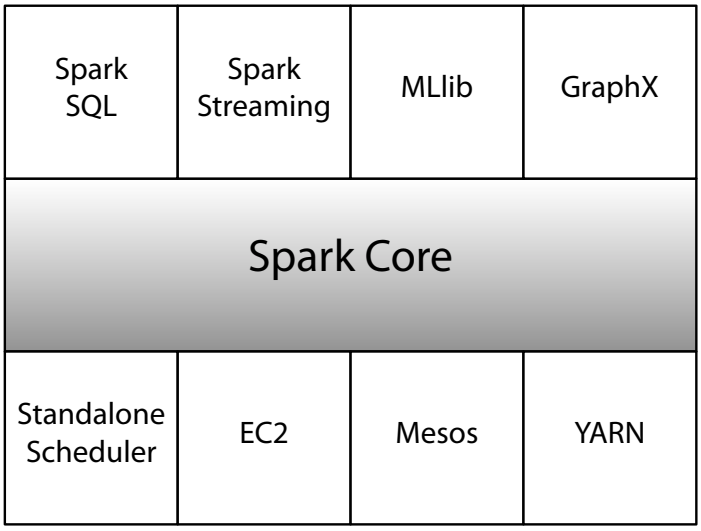


Figure 5.4: Overview Apache Spark Project

5.2.1 Apache Spark

The core of this thesis project is the cluster computing platform *Spark* [26]. Distributed by the Apache Software Foundation with initial development at University of Berkeley, it is the most active big data open source project today and still enjoys a rising popularity. Compared to its predecessor *MapReduce* it is much faster, easier to use with richer APIs and has more functionality such as interactive queries, streaming, machine learning, and graph processing. [27]

As *Spark* is a general engine, it lets the user combine multiple types of computations, such as text-processing, SQL queries and text processing. This makes it a good starting point for new developers, as well as scale existing projects to a huge processing environment. Applications can be programmed in Python, Scala and Java, with 77% of Spark itself written in Scala. [28]

Spark Core is responsible for scheduling, distributing and monitoring applica-

tions across worker machines or a computing cluster. It also contains the API which defines *resilient distributed datasets* (RDDs), which are Spark's basic programming abstraction that defines data across computing nodes.

Spark SQL is the section that defines the work with structured data. It gives functionality to query data via SQL and supports many sources of data, including Hive tabls and JSON.

Spark Streaming allows the processing of live data streams, such as log files, Twitter streams and time series data. It helps with hidden functionality to provide fault tolerance and scalability, just as Spark Core.

MLlib provides common machine learning functions, designed to scale across a computation cluster. Algorithms for classification, regression and clustering are available, as well as additional support through model evaluation and data import.

GraphX is the Spark library for graphs, as in a social network's friend graph. It gives an abstraction layer on top of Spark RDDs to handle parallel computations such as subgraph searches and common algorithms such as PageRank.

YARN and *Mesos* are cluster managers to replace Sparks own *Standalone Scheduler*, for cases where an application extends across many nodes and its configuration needs to be fine-tuned.

Besides the filesystem on a single machine, Spark supports the Hadoop distributed filesystem (HDFS) and others like Amazon S3, Cassandra and Hive.

This project uses the version *1.6.2* of the Spark project, as it fulfils all dependencies and is the latest stable version.

5.2.2 Scala & JVM

Scala is a Java-like programming language which unifies object-oriented and functional programming. Every value is an object, therefore it is a pure object-oriented language. Functions are handled like values, therefore it is a functional language. Nesting of function definitions and higher-order functions are supported.

Scala has been designed to interoperate with Java, as a developer can create Scala classes that call Java methods, inherit from Java class and implement Java interfaces. No glue code or other workarounds are necessary.

The aforementioned flexibility gives the developer a wide selection of tools to choose the one that fits the task best. It provides easy access to all existing libraries in Java and is statically typed, which lets the compiler find and display errors before the program runs.

Scala source code is compiled to Java bytecode, which is executed in a *Java virtual machine* (JVM). This allows the complete interoperability between the two languages, with a lot of open-source projects providing nativ APIs for both.

The created code in this project tries to follow the style described in Code Complete [29]. This project uses the version *2.10.6* of the Scala language.

5.2.3 Kafka & ZooKeeper

A message broker distributes messages from a sender to a receiver in a formal messaging protocol. It mediates communication between applications in

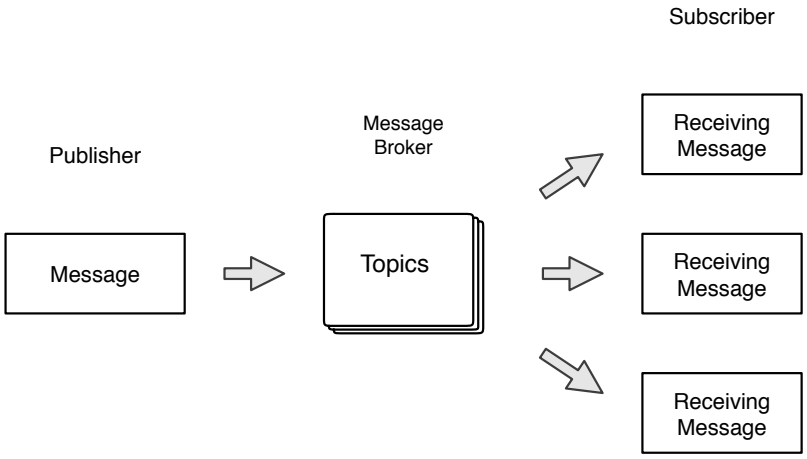


Figure 5.5: Basic message broker configuration

order to decouple them, so that they have no or minimal awareness of each other. A message broker can route, transform, and store messages, as well as respond to errors or other events. Figure 5.5 depicts a simple message broker configuration, based on [30].

They usually work as follows: a publisher creates one or more topics at the central message broker. The publishers task then is to push new messages to one topic. A receiver can subscribe to one or more of those topics and receive the messages that are coming in. The message broker therefore is at the center of the information flow. It needs to handle massive numbers of messages.

Besides the core processing engine *Spark* (see chapter 5.2.1), the message broker is the most important part in the data pipeline. It needs to handle the in- and output in a non-blocking, safe and recoverable fashion. The data needs to be stored redundant in order to preempt any cluster failure.

For these reasons, *Apache Kafka* was chosen as the message broker in this project. Initial development was funded by LinkedIn and Kafka is now used

by companies such as Netflix, PayPal and Spotify. [31] [32] [33] The project is written in Scala (see chapter 5.2.2) and provides a high-throughput, low-latency platform for handling data feeds. Compared to other message broker, like RabbitMQ, it provides ordered delivery and support for simple message re-reads. No complex routing to consumers and message delivery guarantee is needed. Kafka can easily support more than a million packages per second. [34]

Beneath the handling of messages by Kafka, a distributed configuration service is needed. With a cluster manager like *Apache ZooKeeper* more producers and consumers are added to handle an increasing workload and guarantee redundant performance and storage.

5.2.4 DB2Kafka

During development there is no access to a satellite via ground stations. It is desired to implement and test the sending and receiving of messages as well as the algorithm in a computing environment. In order to avoid testing components separately and neglect their interaction, the data has to be streamed through the message broker to the processing engine.

As mentioned in Chapters 5.2.1 and 5.2.3, Spark and Kafka were chosen for these tasks. The data is stored in a MySQL database during development. A way to pull the data out of the database and stream it in chronological order into Kafka was needed. No open-source project existed for this task, therefore I created *DB2Kafka*. The software is written in Python3 [35] using existing open-source projects such as *PyMySQL* and *pykafka*.

The data has to be sent in chronological order. The script converts the relational data in the database into a message for each timestamp, that is then sent to the message broker. The format of these packets is described in Section 5.1. As laid out in Chapter 3, the data is organised in tables for different data types, such as floating point values or unsigned integers. The values for each point in time get collected across these tables. As the further performance is of no interest at this point in the pipeline (see decoupling in Chapter 5.2.3) the data is pushed as fast as possible to the message broker. In order to stream continuously, the program reads in chunks, formats the data into JSON packets and sends them away. Then the cycle is repeated.

5.2.5 PlotMeisterTS

Visualising outliers in a quick way is an important part of the development process (see Section 4.2). As no open-source tool was available for this task, I created PlotMeisterTS. The software is written in Python3 [35] using existing open-source projects such as *PyMySQL* and *matplotlib*. Figure 5.6 depicts the timeseries of PID 977 over the course of close to 63 days. An visible and recognised outlier occurs between 23. and 24. August.

The program expects the PID and point in time where an outlier occurred. It then queries the database for a default time frame around the date. If there are too little data points in this range, it is expanded dynamically. The outlier is plotted when enough data points are available to visually confirm it.

PlotMeisterTS has several features that allow to check for outliers quickly. The processing time is minimised by decreasing the number of queries to

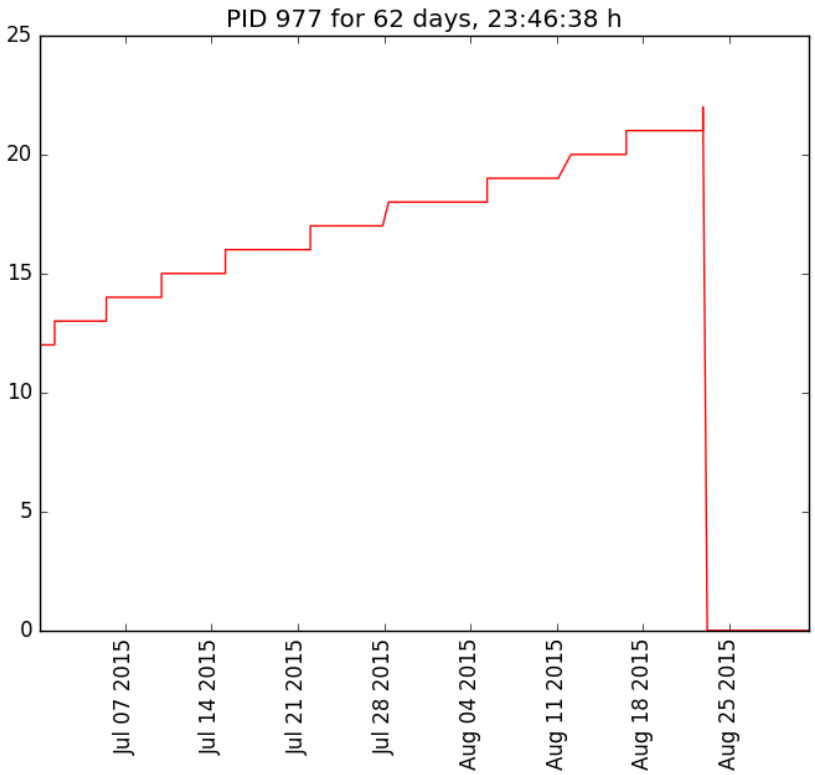


Figure 5.6: Example output of PlotMeisterTS

the database. As the script knows which PID is stored in which table of the database, initially only one request is necessary. If there is too little information in the answer to the first request, e.g. an outlier is not clearly visible, the time period is expanded. Only then multiple queries are required. This happens without additional user interaction, which further decreases the run time of the program.

The current configuration concentrates on the data of this thesis, but it can be easily adapted to other missions. The changes for other missions are made in a configuration file: the minimum and maximum values for a certain PID timeseries have to be adapted, as well as the lists of PIDs that show in which of the database table they are stored. As these values itself can be received by querying the database, the migration to another mission requires little effort.

5.3 Algorithm

5.3.1 SPIRIT

The *Streaming Pattern dIscoveRy in multIple Timeseries* (short *SPIRIT*) is an algorithm developed by researchers at Carnegie Mellon University. [36] *SPIRIT* finds hidden variables, which summarise the key trends in a collection of streaming time series and can recognise novelties due to sudden changes. As it was designed as an algorithm for the live processing of data, it works on a single pass over the data and does not compare each data stream against each other. At any given time SPIRIT provides the number of hidden variables k and the weights $w_{i,j}$. These weights can be seen as the contribution of the i-th

hidden variable to the reconstruction of time series j. Use cases for SPIRIT are the summation of key trends, novelty detection and forecasting.

The benefits of SPIRIT are:

- streaming: it works incremental and is scalable, with memory and processing power independent of the total stream length
- linear scaling: it does not spot correlations by calculating all n^2 pairs
- adaptive: it detects gradual and sudden changes dynamically and determines the number of hidden variables k automatically

In an example system at a time t, all time series possess a periodic pattern under normal operation. SPIRIT discovers the correct number of hidden variables, which in this case is one which describes the periodicity of the pattern. All time series follow this pattern, multiplied by a factor, which is the participation weight w of each observed variable into the hidden variable. If an event outside the normal operation occurs at time $t + 1$, one or more new hidden variables are added, to keep up with time series that are following a chaotic/non-periodic pattern. If normal operation is resumed with all time series in a periodic pattern as at time t, the number of hidden variables returns to one again.

The algorithm is based on PCA (see Chapter 2.7.1). It is adapted to process incoming data in the form of numerical streaming time series. Instead of computing the eigenvectors and eigenvalues from the dataset as a whole, SPIRIT builds the eigenvector incrementally from each incoming part of data.

With a *n* semi-infinite streams producing a value $x_{t,i}$ for every time tick $t = 1, 2, ...$ and every stream $1 \leq i \leq n$, Spirit follows these steps:

- adapting the number hidden variables *k* necessary to summarise the main trends in the data
- adapting the participation weights $w_{i,j}$ of the *i*-th stream on the hidden variable *j* ($1 \leq j \leq n$ and $1 \leq i \leq k$), so as to produce an accurate summary of the stream collection.
- monitoring the hidden variables $y_{t,j}$, for $1 \leq j \leq k$
- updating the variables efficiently

This can be transformed to pseudocode with the given result:

procedure TRACKW

Initialise k hidden variables w_i to unit vectors:

$w_1 = [10 \ldots 0]^T$, $w_2 = [010 \ldots 0]^T$, etc.

Initialise d_i (i=1, $\cdots k$) to a small positive value

As each point x_{t+1} arrives, initialise $\acute{x}_i = x_{t+1}$

for $i \leftarrow 1, k$ **do**

$y_i = w_i^T \acute{x}_i$ ▷ $y_{t+1,i}$ = projection onto w_i

$d_i \leftarrow d_i + y_i^2$ ▷ energy proportional to i-th eigenvalue of $X_t^T X_t$

$e_i = \acute{x}_i - y_i w_i$ ▷ error, $e_i \perp w_i$

$w_i \leftarrow w_i + \frac{1}{d_i} y_i e_i$ ▷ update PC estimate

end for

$\acute{x}_{i+1} = \acute{x}_i - y_i w_i$

end procedure

procedure SPIRIT

Initialise $k \leftarrow 1$

Initialise total energy estimates of x_t and $\tilde{x}_t$ to $E \leftarrow 0$ and $\tilde{E}_1 \leftarrow 0$

while True **do**

$TrackW$ ▷ update w_i for $1 \leq i \leq k$

Update estimates for $1 \leq i \leq k$:

$E \leftarrow \frac{(t-1)E+\|x_t\|^2}{t}$ and $\tilde{E}_i \leftarrow \frac{(t-1)\tilde{E}_i+y_{t,i}^2}{t}$

Estimate of retained energy is $\tilde{E}_{(k)} = \sum_{i=1}^{k} \tilde{E}_i$

if $\tilde{E}_{(k)} < f_E E$ **then**

Initialise k hidden variables w_i to unit vectors:

$w_1 = [10 \ldots 0]^T$, $w_2 = [010 \ldots 0]^T$, etc.

Initialise $\tilde{E}_{k+1} \leftarrow 0$

Increase $k \leftarrow k + 1$

else if $\tilde{E}_{(k)} > F_E E$ **then**

Discard w_k and $\tilde{E}_k$

Decrease $k \leftarrow k - 1$

end if

end while

end procedure

The energy E_t of the sequence x_t is formally defined as:

$$E_t = \frac{1}{t}\sum_{\tau=1}^{t} \|x_\tau\|^2 = \frac{1}{t}\sum_{\tau=1}^{t}\sum_{i=1}^{n} x_{\tau,i}^2 \tag{5.1}$$

f_E and F_E are low-energy and high-energy thresholds. The reconstruction $\tilde{x}_t$ of the algorithm therefore retains the portion of x_t between these two levels. According to recommendations in the literature [17] we use a lower energy threshold $f_E = 0.95$ and an upper energy threshold $F_E = 0.98$.

The novelty detection algorithm in this thesis project is based on the SPIRIT

algorithm. The original description by Papadimitriou et al. and a previous reference implementation in Matlab code [37] was given, the used SPIRIT algorithm is implemented and extended in Scala and Java.

Ill-performing functions provided by the programming languages, such as matrix rotations, are replaced by open-source libraries.

Colt by CERN [38] is a library for high performance scientific and technical computing in Java, which provides fast matrix decomposition functionality. Colt gave a larger speedup with less memory usage, compared to JAMA [39], a Java matrix package, and EJML [40], a linear algebra library for Java.

The key step of the algorithm, as described above, is the principal component analysis (PCA). In order to leverage the mentioned benefits of distributed computing, PCA has to be compatible with this environment.

The basic implementation of PCA works when all of the data is available in one place and fits into the available memory. The core algorithm relies on matrix operations. The data in this project consists of thousands of time series and can be thought of as one data matrix. In order to speed up the process of dimension reduction, we need to find a way to distribute the PCA algorithm, without relying on sending the total amount of data to every node.

This problem is solved, by following the instructions of Qu et al. in [41] with the title 'Principal component analysis for dimension reduction in massive distributed data sets'. It proposes a new method for computing a global principal component analysis (PCA) for the purpose of dimension reduction in data distributed across several locations. The approach given in the paper is to perform local PCA on local data without data moving between computing

nodes. Only after local computation is done, the local PCA results are moved to a central location and merged into a global PCA. The representation of local data by a few local principal components greatly reduces data transfers with minimal degradation in accuracy.

It reduces the communication between nodes, but introduces approximation errors. The tradeoff is, that we introduce a small amount of approximation for which we gain the omission of data transfers and synchronisation. For this algorithm the central computing node or master can become the bottleneck of computation and computations, but this is a problem we already introduced by using Spark.

Each node has two requirements, of which at least one has to be met:
k, the number of local principal components, or α, the minimum local variation. The descriptive statistics at each location are:

- n_i: the number of observations at the the i^{th} location
- $\bar{x}_i$: the vector of column means of i^{th} data set
- $\tilde{\Lambda}_i$: diagonal matrix containing the k_i largest eigenvalues
- $\tilde{U}_i$: matrix whose columns are the k_i eigenvectors corresponding to the k_i eigenvalues in $\tilde{\Lambda}_i$

The following steps describe the core distributed PCA algorithm:

1. set local requirements α and k and receive local dataset
2. compute PCA locally

3. communicate descriptive statistics back to master node

4. compute the data covariance matrix **S** with

$$n\mathbf{S} = X^T(\mathbf{I} - n^{-1}\mathbf{1}\mathbf{1}^T)\mathbf{X} = X^TX - n^{-1}\bar{x}\bar{x}^T \tag{5.2}$$

5. compute principal components $n\tilde{S} = UDU^T$

6. compute global dimensionally reduced representation of the data with $Z = (X_i - \bar{x}^T)U$

Speed gains are expected by merging the streaming algorithm Spirit with the distributed PCA algorithm. Details on testing and evaluation follow in the next chapter.

Chapter 6

Evaluation

This chapter explains the metrics used for comparing the implementations and gives their results. The performances of the new Spirit-on-Spark system, created during this thesis project, versus the existing Java implementation, which is based on a previous Matlab codebase, are compared.

6.1 Benchmark

The following criteria are based on the requirements overview in Table 4.1. The basic requirement for any algorithm is the proper detection of outliers in the data. The outliers given as output by an algorithm can be visually confirmed by using the PlotMeisterTS tool, see 5.2.5.

As discussed in Chapter 4, the purpose of this project is to operate the algorithm in a streaming mode, where data is delivered live from a satellite in orbit. Related to this point is the desire to have the algorithm ready for a production

system, where the input funnel might be swapped, but the remaining system stays the same. (see Chapter 4.2)

The major metric for usefulness of a system in live data analysis is the throughput, given by objects processed per second. An object represents a package of all data which was recorded at one specific timestamp across all available time series. If all underlying basic requirements are met, the performance of an algorithm is determined by this metric.

6.2 Setup

In order to make a meaningful comparison of the performance, we limit the number of time series to 300. In several test runs, the average throughput was measured until it converged against a specific value. This was repeated several times on the same computer, to lessen the effect of other running processes and their fluctuating memory usage.

The test computer is an Apple laptop (model MacBookPro12,1) with a 2.7 GHz Intel Core i5 processor and 16 GB 1867 MHz DDR3 memory. The offline data in the SQL database is stored on an external USB3 SSD (model Samsung MU-PT250B).

The messaging environment created by Zookeeper and Kafka runs on the same test computer, to minimise the effect of external network issues and other problems not concerning the algorithms. This setup might differ from a future production system, but gives close to optimal results which is desired in this comparison.

The code is executed with the *spark-submit* command and runs in the JVM. In each benchmark run all of the data as described in Chapter 3 is processed by the system.

6.3 Result

In this section I will discussed the results of the benchmarks explained above.

The answer to the basic requirements posed in the first section of this chapter are given in the following table 6.1

Table 6.1: Evaluated requirements

Requirement	Java Spirit	Spirit-On-Spark
Outlier Detection	YES	YES
Streaming Operation	NO	YES
Distributed Computing	NO	YES
Production System	NO	YES

As described in Chapter 5, the Spirit-On-Spark algorithm was designed with the streaming mode of operation and the move to a production system in mind. This is opposed to the basic Java implementation, which proves the functionality in an object-oriented language, as a port from earlier Matlab code. Both approaches fulfil the outlier detection, as the core remains the same.

6.4 Number of novelties

There is a proprietary novelty detection running at ESA previous to this thesis project. It works as an offline analysis with fixed limits. We let both approaches analyse the same data, as described in Chapter 3, and then compare the results.

Table 6.2: Algorithm result comparison

Novelty Detection	ESA NovDect	Spirit-On-Spark
Nr. of novelties	98	120
Streaming Operation	NO	YES

A desired outcome of the system is to get more possible novelty candiates, which can then be quickly confirmed by Mission Operators on-site. Therefore a higher number in the given test data is expected.

6.5 Throughput

Comparing the throughput from several test runs we get:

Java Spirit **53 obj/s** (std. deviation 6)
Spirit-on-Spark **405 obj/s** (std. deviation 113)

This shows a speedup of 840% for Spirit-on-Spark, compared to the previous implementation.

These results only give the relative speed gain. The time constraint needs to be tested as well. Is it possible to process the data created during one orbit in less

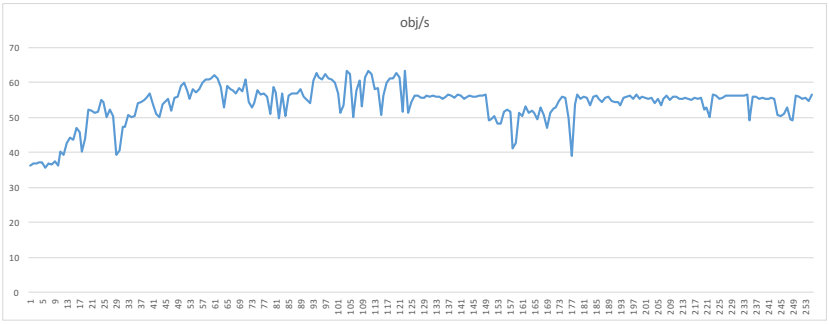


Figure 6.1: Throughput Java Implementation

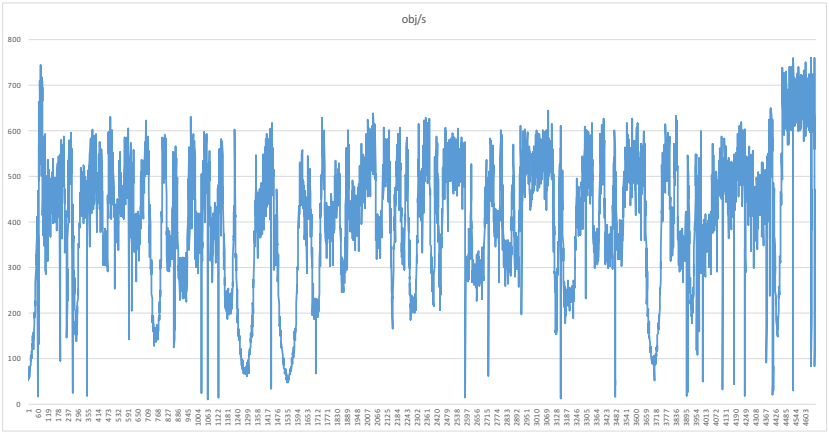


Figure 6.2: Throughput Spirit-on-Spark Implementation

than 15 minutes? (see Chapter 4 for details on this requirement)

The amount of data that a satellite produces during on orbit or between two connections to a ground station needs to be estimated. Given the historical data as described in Chapter 3, 300 PIDs have 191.205 data points in one orbit.

The above performance is 405 objects processed per second and the time duration is 15 minutes. This gives the rough estimation of 364.500 on average, with 262.800 as worst-case and 466.200 as best-case performance, within one standard deviation.

The system is able to process the amount of data in the required time frame, as 191.205 data points can be easily processed in a hard time limit of 15 minutes. Even under worst-case performance the average amount of data can be easily processed. There is enough performance buffer included in the system to handle short peaks of larger amounts of data coming in through the pipeline.

Chapter 7

Conclusion

From the initial problem scenario and the given environment, the objectives for this thesis can be summarised as: Design a system which is able to detect outliers in streaming telemetry data, especially in the time-constraint of a single overpass. The defined requirements are: the input as telemetry time series, a streaming mode of operation and an algorithm that leverages distributed computing. All of these challenges were met.

This thesis proposes a solution based on distributed computing and the SPIRIT algorithm. It merges the best of the SPIRIT algorithm with a distributed PCA method to improve performance. The initial goal, to speed up the outlier processing, was fulfilled. Fast live processing during one orbit of an Earth-observing satellite is available. Expecting a 30 minute window of communication, the data from one orbit can be analysed in 15 minutes. The second half of the window can therefore be used by human operators to plan and execute actions based on the analysis result.

The project provides an 8x speed improvement compared to previous imple-

mentations. The new *Spirit-on-Spark* is faster, ready for production, and easily expandable. The SPIRIT algorithm can now be leverage in a distributed computing environment.

An original contribution of this thesis is the new multi-threaded, distributed computing algorithm and environment for outlier detection algorithms. The functionality is proven with the SPIRIT algorithm. The provided work in this thesis is a combination of research in distributed computing and implementation of these findings. The computing environment facilitates simple comparison of a host of different approaches to novelty detection. It is highly modular and can still be used to compare performances in fixed benchmarks.

The two different setups, development and production, are additional results of this project. These environments are the most fitting for their respective tasks. At the core they are similar with message broker and computing cluster, but differ in inputs and outputs.

In which ways this thesis can be the basis for future work is explained in the next chapter.

Chapter 8

Future work

The presented work has the potential to be expanded in several directions.

A straightforward extension is to move the current setup to a production system, with a proper message broker on a separate server, several topics and subscribers for multiple missions. This requires work in the move to a new environment, as well as setup and configuration work. Concepts for data redundancy and the storage of outlier information could be developed.

Another direction is to expand the computing cluster across several servers and to connect those with a high-speed network. This would provide more computing power for faster data processing. With a multi-purpose computing farm, several missions could be provided with real-time outlier information. The opposite approach, servers dedicated to single missions, would result in unused resources, which is not recommended.

After outliers are detected they need to be analysed by human operators. This process could be sped up by using projects such as ElasticSearch [42], a dis-

tributed search engine and Hadoop [43], a framework for storing data on large clusters.

Another interesting route to expand this thesis is to combine new techniques with the SPIRIT algorithm, such as distributed pattern recognition given in [44]. This requires a major extension of the work, which would then be able to fully leverage the nature of the distributed computing environment.

Another path is to replace *SPIRIT* altogether with another type of detection technique. Candidates for this replacement would be semi-supervised learning algorithms, that can be trained ahead of the satellite deployment, to facilitate improved accuracy and speed. The training and the utilisation in production has to be of concern in the spacecraft design process before launch. Therefore it is an endeavour for future Space missions, which can build on the work of this thesis.

References

[1] Robert H. Shumway DSS. Time series analysis and its applications. 3rd ed. Springer; 2011.

[2] Esling P, Agon C. Time-series data mining. ACM Comput Surv 2012;45:12:1–12:34. Available at: http://doi.acm.org/10.1145/2379776.2379788.

[3] Shatkay H, Zdonik SB. Approximate queries and representations for large data sequences. In:. Proceedings of the twelfth international conference on data engineering, Washington, DC, USA: IEEE Computer Society; 1996, pp. 536–45. Available at: http://dl.acm.org/citation.cfm?id=645481.653263.

[4] Oliveira ALI, Lemos Meira SR de. Detecting novelties in time series through neural networks forecasting with robust confidence intervals. Neurocomputing 2006;70:79–92. Available at: http://dx.doi.org/10.1016/j.neucom.2006.05.008.

[5] V. Barnett TL. Outliers in statistical data 3rd edition 1994:584 pp.

[6] Ryan J, Ryan J, Lin M-j, Miikkulainen R. Intrusion detection with neural networks. ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 1998;10:943–9.

[7] Aggarwal CC. Outlier analysis. Springer; 2013. Available at: http://dx.doi.org/10.1007/978-1-4614-6396-2.

[8] Markou M, Singh S. Novelty detection: A review - part 1: Statistical approaches. Signal Process 2003;83:2481–97. Available at: http://dx.doi.org/10.1016/

j.sigpro.2003.07.018.

[9] Hodge VJ, Austin J. A survey of outlier detection methodologies. Artificial Intelligence Review 2004;22:2004.

[10] Koskivaara E. Artificial neural network models for predicting patterns in auditing monthly balances. Journal of the Operational Research Society 2000;51:1060–9. Available at: http://dx.doi.org/10.1057/palgrave.jors.2601014.

[11] Pimentel MAF, Clifton DA, Clifton L, Tarassenko L. Review: A review of novelty detection. Signal Process 2014;99:215–49. Available at: http://dx.doi.org/10.1016/j.sigpro.2013.12.026.

[12] Sola J, Sevilla J. Importance of input data normalization for the application of neural networks to complex industrial problems. IEEE Transactions on Nuclear Science 1997;44:1464–8.

[13] Forman G. BNS feature scaling: An improved representation over tf-idf for svm text classification. In:. Proceedings of the 17th acm conference on information and knowledge management, New York, NY, USA: ACM; 2008, pp. 263–70. Available at: http://doi.acm.org/10.1145/1458082.1458119.

[14] Aksoy S, Haralick RM. Feature normalization and likelihood-based similarity measures for image retrieval. Pattern Recogn Lett 2001;22:563–82. Available at: http://dx.doi.org/10.1016/S0167-8655(00)00112-4.

[15] Lee JA, Verleysen M. Nonlinear dimensionality reduction. 1st ed. Springer Publishing Company, Incorporated; 2007.

[16] Fu T-c. A review on time series data mining. Engineering Applications of Artificial Intelligence 2011;24:164–81. Available at: http://www.sciencedirect.com/science/article/pii/S0952197610001727.

[17] Jolliffe I. Principal component analysis. Springer Verlag; 2002.

[18] University TPS. STAT 897D: Applied data mining and statistical learning n.d. Avail-

able at: https://onlinecourses.science.psu.edu/stat857/node/11.

[19] Coulouris G, Dollimore J, Kindberg T, Blair G. Distributed systems: Concepts and design. 5th ed. USA: Addison-Wesley Publishing Company; 2011.

[20] 2016. Available at: http://www.ntp.org/.

[21] Gupta M. AC Gao J. Outlier detection for temporal data - a tutorial n.d. Available at: https://www.siam.org/meetings/sdm13/gupta.pdf.

[22] Solenix 2016. Available at: http://solenix.ch/.

[23] ESA 2016. Available at: http://www.esa.int/ESA.

[24] Data pipeline - best practices 2016. Available at: http://radar.oreilly.com/2015/09/three-best-practices-for-building-successful-data-pipelines.html.

[25] DTAP—Development, testing, acceptance, and production 2016. Available at: https://www.phparch.com/2009/07/professional-programming-dtap-%e2%80%93-part-1-what-is-dtap/.

[26] Apache spark 2016. Available at: https://spark.apache.org/.

[27] Karau H, Konwinski A, Wendell P, Zaharia M. Learning spark: Lightning-fast big data analytics. 1st ed. O'Reilly Media, Inc. 2015.

[28] Apache spark code repo 2016. Available at: https://github.com/apache/spark/.

[29] McConnell S. Code complete, second edition. Redmond, WA, USA: Microsoft Press; 2004.

[30] Message broker - working with topics 2016. Available at: https://docs.wso2.com/display/MB300/Working+with+Topics.

[31] Kafka at netflix 2016. Available at: http://www.slideshare.net/wangxia5/netflix-kafka.

[32] Kafka at paypal 2016. Available at: http://www.slideshare.net/Couchbase/paypal-creating-a-central-data-backbone-couchbase-to-couchbase-to-kafka-

to-hadoop-and-back-couchbase-connect-2015.

[33] Kafka at spotify 2016. Available at: https://labs.spotify.com/2016/02/25/spotifys-event-delivery-the-road-to-the-cloud-part-i/.

[34] n.d. Available at: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[35] Python3 2016. Available at: https://docs.python.org/3/.

[36] Papadimitriou S, Papadimitriou S, Sun J, Faloutsos C. Streaming pattern discovery in multiple time-series. IN VLDB 2005:697–708.

[37] Matlab 2016. Available at: http://mathworks.com/products/matlab/.

[38] Colt 2016. Available at: https://dst.lbl.gov/ACSSoftware/colt/.

[39] JAMA: A java matrix package 2016. Available at: http://math.nist.gov/javanumerics/jama/.

[40] EJML 2016. Available at: http://ejml.org/wiki/index.php?title=Main_Page.

[41] Qu Y, Ostrouchov G, Samatova N, Geist A. Principal component analysis for dimension reduction in massive distributed data sets. In:, 2002.

[42] ElasticSearch 2016. Available at: https://www.elastic.co/products/elasticsearch.

[43] Hadoop 2016. Available at: https://hadoop.apache.org/.

[44] Sun J, Papadimitriou S, Faloutsos C. Distributed pattern discovery in multiple streams. In: Ng W-K, Kitsuregawa M, Li J, Chang K, editors. Advances in knowledge discovery and data mining: 10th pacific-asia conference, pakdd 2006, singapore, april 9-12, 2006. proceedings, Berlin, Heidelberg: Springer Berlin Heidelberg; 2006, pp. 713–8. Available at: http://dx.doi.org/10.1007/11731139_82.