DATA MINING TOOLS
IN SUPPORT OF
SOFTWARE TESTING

Thesis

Written by:

Benkő Beatrix
Bachelor of Science in Mathematics
Mathematical Analyst specialisation

Supervisor:

Lukács András
Institute of Mathematics
Department of Computer Science

Eötvös Loránd University
Faculty of Science
Budapest, 2017
ABSTRACT

The aim of this thesis was to find data mining tools to support the software testing process, specifically the endurance test result evaluation of IP Multimedia Subsystem applications. In order to improve the test result evaluation process based on the data from the previous test results, classification and anomaly detection algorithms have been applied.

Supervised classification algorithms have been applied using the test result annotation of software testers. The correctness of annotation was assumed in case of this approach. Unsupervised anomaly detection algorithms have been applied in order to be able to identify the anomalous test results without relying on the annotation.

Two state-of-the-art models, the supervised Extreme Gradient Boosting and the unsupervised Isolation Forest, have proved to perform well, they offer a reliable solution to the proposed data mining task.

This thesis covers the endurance test result evaluation only, but because of the similarity between the different performance tests, the achieved results could be generalised. Further optimisation of the suggested methods and testing them in real-life situation may be part of future work.
# TABLE OF CONTENTS

Abstract ................................................................. I

Table of contents ..................................................... II

List of figures ........................................................ IV

List of tables .......................................................... V

1 Introduction .......................................................... 1

2 Software testing background ..................................... 3
   2.1 Performance testing ............................................ 4
   2.2 Endurance tests ................................................ 4
      2.2.1 Endurance testing of an IMS application ............... 5

3 Data mining methodology .......................................... 7
   3.1 Process of data mining ....................................... 7
   3.2 Data mining tasks ............................................. 7
      3.2.1 Predictive modelling .................................... 8
      3.2.2 Anomaly detection ..................................... 9

4 Formulation of proper data mining task ....................... 10

5 Development environment ......................................... 12
   5.1 Environment .................................................. 12
   5.2 Programming language and libraries ....................... 12

6 Data sets .................................................................. 14

7 Applied data mining models ...................................... 20
   7.1 Supervised models ............................................. 20
      7.1.1 Naive Bayes .............................................. 21
      7.1.2 Decision trees and ensemble methods ................ 22
         7.1.2.1 Random Forests .................................... 24
         7.1.2.2 Adaptive Boosting ................................ 25
         7.1.2.3 Extreme Gradient Boosting ....................... 26
   7.2 Unsupervised models ......................................... 26
      7.2.1 One-Class Support Vector Machine ................. 27
8 Application of models and evaluation ................................. 30
  8.1 Supervised models ................................................... 30
      8.1.1 Naive Bayes .................................................. 31
      8.1.2 Random Forest ............................................... 32
      8.1.3 Adaptive Boosting ........................................... 34
      8.1.4 Extreme Gradient Boosting ................................. 35
      8.1.5 Evaluation of supervised models ......................... 37
  8.2 Unsupervised models .................................................. 40
      8.2.1 One-Class Support Vector Machine ....................... 40
      8.2.2 Isolation Forest ............................................. 42
      8.2.3 Evaluation of unsupervised models ................. 44

9 Decision path .......................................................... 46

10 Conclusion ............................................................... 49

References ................................................................. VI
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Part of an XML file</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>Histogram - number of NaN values</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>Histogram - number of zeros</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Histogram - number of positive values</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Histogram - number of ids for each parameter</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>NB - ROC curves</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>NB - Confusion matrices</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>RF - ROC curves</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td>RF - Confusion matrices</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>AB - ROC curves</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>AB - Confusion matrices</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>XGB - ROC curves</td>
<td>36</td>
</tr>
<tr>
<td>13</td>
<td>XGB - Confusion matrices</td>
<td>37</td>
</tr>
<tr>
<td>14</td>
<td>AUC box-and-whisker diagram - NB</td>
<td>38</td>
</tr>
<tr>
<td>15</td>
<td>AUC box-and-whisker diagram - RF, AB and XGB</td>
<td>38</td>
</tr>
<tr>
<td>16</td>
<td>OCSVM - ROC curves</td>
<td>41</td>
</tr>
<tr>
<td>17</td>
<td>OCSVM - Confusion matrices</td>
<td>42</td>
</tr>
<tr>
<td>18</td>
<td>IF - ROC curves</td>
<td>43</td>
</tr>
<tr>
<td>19</td>
<td>IF - Confusion matrices</td>
<td>43</td>
</tr>
<tr>
<td>20</td>
<td>IT</td>
<td>47</td>
</tr>
<tr>
<td>21</td>
<td>XGB - CART</td>
<td>48</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

1. Used software and their version ......................................... 13
2. Number of test runs and number of XML files ..................... 14
3. Mean and variance of AUC .................................................. 37
4. Confidence intervals of AUC ............................................... 39
5. Training, evaluation and total run time of supervised models .... 39
6. AUC of unsupervised models .............................................. 44
7. Training, evaluation and total runtime of unsupervised models . 44
1 Introduction

As John Naisbitt stated, literally "We are drowning in information but starved for knowledge". Living in the information age means that nowadays we are able to collect, store, transfer tremendous volumes of data easier than ever. Meanwhile, many areas exist, where an excessive amount of data is being accumulated, but the hidden information in it is not exploited. However, extracting valuable information has proven to be extremely challenging, mostly because traditional data analysis tools and techniques cannot be used due to the massive size of data sets.

And here lies the vast importance of data mining: the ability to automatically find useful information in large amounts of data and make use of it by transforming it into valuable knowledge, achieving this trough blending traditional data analysis methods with sophisticated algorithms for processing large volumes of data. In fact, data mining was built upon previously used methodologies and algorithms, drawing upon statistics -sampling, estimation, hypothesis testing-, artificial intelligence, machine learning, pattern recognition, and adopting techniques from optimisation, visualisation, information retrieval, database systems, high performance computing. The versatility of the field and the effective combination of different techniques enabled us to develop more powerful and scalable tools, capable of accessing, analysing, and finding patterns in Big Data in a way that is beyond human capabilities. Thus, we can automate and prioritise routine decision making processes, achieve better outcomes sooner, and help to eliminate human errors.

One field, where such tools could enhance the human work, is software testing. During my internship at a software testing team, I realised that despite they are trying to automate the testing process, the test result evaluation part still requires their domain knowledge and it is the most time consuming phase, also human testers easily can omit observing faulty behaviour of the software. Data mining can be one means to the improvement of this phase. The aim of my thesis was to find tools of data mining, which could help and quicken the work of software testers, specifically the evaluation of the test results by utilising a large amount of available data and trying to eliminate at least a part of human assistance.

The rest of this thesis is organised as follows. The first two sections offer an insight into performance software testing and respectively data mining, with regard to those who are not familiar with these two topics. I represent the process and purpose of stability tests in section 2 and summarise the essential data mining concepts and techniques related to my work in section 3. Additionally, in section 4, I point out how test result evaluation can be converted into a data mining problem. In the ensuing section I provide a short description of the development environment,
as the title of section 5 suggests. In section 6 are presented the initial data sets, the
applied data transformation methods, and the two final data sets on which the data
mining models were tested. Consequently, in the subsequent section 7 is described
the mathematical background of the applied data mining models. In section 8 is
discussed how the models were applied, performance of each model on the data
sets, and they are compared and evaluated based on different metrics. In section 9
I present how the decision of a certain model can be used to support the decision
making in the test result evaluation process. Finally in the last section I outline
the achieved results of the research process and summarise these besides suggesting
further relevant improvements in section 10.
2 Software testing background

In this introductory section related to software testing I intend to provide the readers with an overall view of the topic. The section is based on [1] and [2] sources, which provide further information for those who are interested.

Basically, software testing is an investigation process conducted to provide information about the quality of the software. The main objective is to determine how well the evaluated software conforms to its specifications, besides regaining reliability after modifications of the software.

There are several methods available in software testing, the most common of them are the following ones: static and dynamic, white-box and black-box testing. Static testing refers to review, walkthrough, or inspection of the source code itself, thus it is called verification, whereas dynamic testing indicates actual execution of the programmed code with test cases, hence it is designated as validation. Regarding the box-approach, while the white-box approach uses the actual code of the tested program to perform analysis, the black-box approach compares the program input against the output without taking into account inner workings. Concerning the different test types, I outline here the two major categories: functional and non-functional testing. Functional testing, as its name suggests, it tests a particular action or functionality of the software, whilst non-functional testing refers to testing quality characteristics and the way a software operates and involves testing aspects such as scalability or other performance, behaviour under certain constraints, or security. Both testing types are dynamic testings and typically are done using the black-box approach.

The software testing process may follow different development models, such as the traditional waterfall model, V-model, agile model, the most commonly utilised being the last two of them. Each process model follows a particular life cycle, I do not detail them here, but they are well described in the [2] source. The testing process itself can be divided into five major steps: planning and control, analysis and design, implementation and execution, evaluating exit criteria and reporting, and finally closure. Traditionally, all five phases required human testers. Nowadays, automated software testing methods are taking over the role of humans, but these methods still must be ameliorated.

This thesis focuses on the improvement of endurance test result evaluation, so the following subsections aim to enlighten readers on the topics of performance testing and endurance tests.
2.1 Performance testing

Software performance testing is considered a type of non-functional testing, and it is performed with the intention of determining how a software performs in terms of responsiveness and stability under a particular workload. It can also contribute to measurement or verification of other quality attributes of the software too, such as scalability, reliability and resource usage. In fact, performance testing can serve the following purposes:

- demonstrate that the software meets performance criteria
- compare two software to find out which one performs better
- measure which parts of the software cause it to perform poorly.

Types of performance testing include stress tests, load tests, spike tests, configuration tests and isolation tests, and last but not least, endurance tests. Stress testing is done to determine the software’s robustness in terms of extreme load and helps to predict whether the software will perform sufficiently if the load goes above the expected maximum. Load tests are conducted to understand the behaviour of the application under a specific expected load, thus helping to identify the maximum operating capacity of it, as well as the bottlenecks. Spike testing is done by suddenly increasing or decreasing the load, and observing the behaviour of the software, leading to understanding whether the software will be able to handle dramatic changes in load. Configuration tests are performed to determine the effects of configuration changes to the software’s components on the software’s performance and behaviour. Isolation testing involves repeating a test execution that resulted in a software failure or crash. Such testing can help to isolate and confirm the fault domain.

In reference to endurance testing, the next subsection provides the readers with sufficient information. Also, I would like to highlight here the simple observation that these types of performance software testing basically are very similar to each other, thus generalisation from the results presented in this thesis related to endurance testing should be easy at all.

2.2 Endurance tests

Endurance testing, also known as soak testing, is a non-functional type of software testing, one of the simplest forms of performance testing. Its results can help to verify if the software can sustain the continuous expected load without performance degradation.
This type of testing implies testing a software with a significant load extended over a significant period of time, to discover how it behaves under sustained use. For example, a software may behave exactly as expected when tested for one hour, but when the same software is tested for three hours, problems such as memory leaks already can cause the system to fail or behave randomly. Thus, memory usage should be monitored, as well as other possible sources of failure in order to detect such potential leaks. Also, it should be considered that longer tests, for instance a weekend long one, could reveal even more problems of such type. If it is not possible to conduct such an extended test, it may be required to extrapolate the test results. As an illustration, if the software is required to process ten thousand transactions over one hundred hours, it may be possible to complete processing the same ten thousand transactions in a weekend-long duration as representative sample of the actual use. A good soak test should also include the ability to simulate peak loads as opposed to just average loads. If manipulating the load over specific periods of time is not possible, alternative solution can be allowing the system to run at peak production load for the duration of the test.

Hence, endurance tests should be run under controlled conditions, to accurately measure the capabilities of the software. The process involves simulating real-life environment and user load for the target software. This method helps to determine how it behaves when it needs to serve multiple users simultaneously. Observations and measurements made during test could be used to improve the characteristics of the software under test.

All in all, endurance tests are used primarily to discover how the software behaves under sustained use. In this way, it is ensured that software performance after some long period of sustained activity is as good or better than at the beginning of the test.

In the following subsection I describe in more detail how it comes about the endurance testing of an application software in practice.

### 2.2.1 Endurance testing of an IMS application

Hereinafter, I describe more specifically the endurance test of an IP Multimedia Subsystem (IMS) application, which can be part of a telecommunications network. The main concept here remains the same: testing the application at a significant load extended over a significant period of time, to discover how it behaves under sustained use.

Tests are launched by the software testers. During test run, a data collector tool measures a set of parameters at regular time intervals, to be more specific, every
fifteen minutes. The measured parameters, the hardware locations of measurements and the measured values are stored in documents encoded in Extensible Markup Language (XML). The parameters characterise the behaviour of the application, the expectation is that the measured values of these parameters will be approximately constant or consistently increasing during the whole duration of a single test run. After the test run has been finished, software testers have to evaluate the test results. They visually check automatically generated diagrams and aggregated values of the data stored in XML files, looking for anomalies, unusual values which would indicate software failure.

From all of this it follows that in the case of an IMS application the test result evaluation phase requires domain knowledge and can be time consuming. In addition, if testers are not open-eyed, they can easily omit observing faulty behaviour of the software which has caused only a small but significant change in the measured values and that has been further diminished during aggregation. In order to overcome these problems, one improvement could be to make use of the large amount of data stored in XML files, and apply data mining algorithms to transform the hidden information in raw data into valuable knowledge. In this way, it may be possible to eliminate at least a part of human assistance in routine decision making too, or at least support it with the knowledge extracted from the available data.

The key to understanding how data mining could be applied in this case is the comprehension of data mining process and core data mining tasks, which i am going to present in the following section, regarding those who lack information related to the topic.
3 Data mining methodology

In this section I begin with the essence of data mining process and a discussion of two core data mining tasks. The main sources of information provided here were [3], [4] and [5], which are excellent sources to acquire deeper understanding of the topic.

3.1 Process of data mining

As mentioned in section 1, data mining is an interdisciplinary field involving methods of machine learning, statistics, database systems, and many other fields. Consequently, it hardly can be circumscribed, it is difficult to find a complete definition of it because of its complexity. Despite this fact, Gregory Piatetsky-Shapiro grasps the essence of data mining, defining the process itself as "The nontrivial extraction of implicit, previously unknown, and potentially useful information from data" [6]. This interpretation encapsulates the main objective of data mining: valuable, unknown information retrieval, data-driven pattern discovery and anomaly detection in massive data sets.

In this way, data mining can be perceived as a process of knowledge discovery, which usually is the result of finding previously unknown, hidden information. It forms an integral part, but only a step in the notorious process of Knowledge Discovery in Databases (KDD), which is the overall process of converting raw data into useful information.

In order to apply the KDD in case of industrial projects, the CRoss-Industry Standard Process for Data Mining (CRISP-DM) has been introduced by Pete Chapman [7]. My work related to this thesis also followed similar process phases.

The modelling phase of CRISP-DM usually is based on which of the core data mining tasks must be accomplished. In the following subsections, I shortly present two of the core data mining tasks, according to [3]. There are several algorithms which could be applied in each case, I am going to mention the most important ones for both types of task.

3.2 Data mining tasks

As described both in the [3] and [4] sources, we can demarcate two major categories of data mining tasks: descriptive and predictive tasks.

Descriptive mining tasks characterise properties of the data in a target data set. The scope of these tasks is to derive patterns, correlations, trends, anomalies, that summarise the underlying relationships in data. Descriptive data mining tasks
are often exploratory in nature and frequently require post-processing techniques to validate and explain the results.

Predictive mining tasks perform induction on the current data in order to make predictions. The objective here is to predict the value of a particular attribute based on the values of other attributes. The attribute to be predicted is commonly known as the target or dependent variable, while the attributes used for making the prediction are known as the explanatory or independent variables.

In this thesis, firstly I have devoted effort on predictive mining tasks, as my main goal was to categorise new test run results as okay or not okay tests based on previous test run results. But descriptive tasks, such as anomaly detection also have been applied, as their results could offer support information during the decision making process of human testers.

Based on the source [3], we can distinguish four core data mining tasks, two of them are presented in detail in the following separate subsections. The mentioned algorithms are well described in the [3] source, additionally there are provided examples for better understanding.

### 3.2.1 Predictive modelling

Predictive modelling refers to the task of forecasting the value of the target variable as a function of the explanatory variables with the use of an adequate model. The goal of predictive modelling always is to create a model which minimises the error between the predicted and real value of the target variable. Usually two types of these tasks are distinguished: in case of discrete target variables classification is used, and for continuous target variables regression is applied. For instance, predicting whether a test run was okay or not is a classification task because the target variable is binary-valued. On the other hand, forecasting the future values of a measured parameter during a test run is a regression task because measured values are continuous-valued.

One simple algorithm offering a solution to such tasks is the K-Nearest Neighbours algorithm, which predicts the target variable as a function of its neighbours based on a similarity measurement, for example, the euclidean distance. Other approaches, such as the Naive Bayes models, Decision Trees, Support Vector Machines, Artificial Neural Networks also can be applied in case of predictive modelling tasks.

With the use of these approaches, real-life applications of predictive modelling can be solved: identifying customers that will respond to a marketing campaign, predicting disturbances in the Earth’s ecosystem, or judging whether a patient has a particular disease based on the results of medical tests.
3.2.2 Anomaly detection

Anomaly detection covers the task of identifying observations whose characteristics are significantly different from the rest of the data. Such observations are known as anomalies or outliers. The goal of an anomaly detection algorithm is to discover the real anomalies and avoid falsely labelling normal objects as anomalous. In other words, a good anomaly detector must have a high detection rate and a low false alarm rate.

Algorithms able to perform anomaly detection include outlier detection based on K Nearest Neighbours, One-Class Support Vector Machines, Replicator Neural Networks or Isolation Forests. In fact, the classification algorithms hardly can be delimited from the ones used for outlier detection, as the algorithms used for predictive modelling also can be applied to anomaly detection tasks.

Some serious problems can be resolved by applying anomaly detection algorithms, these include: credit card fraud detection, detection of network intrusions, discovering unusual patterns of disease and also ecosystem disturbances.

These two data mining tasks represent the two approaches based on which I tried to offer a solution to improve the endurance test result evaluation.

In the following subsection is presented how the test result evaluation improvement could be formulated as adequate data mining task.
4 Formulation of proper data mining task

After the previous two sections, from now on I assume that the readers have familiarised themselves with both the software testing and data mining fundamentals required for the comprehension of the problem and the solution approaches proposed in this thesis.

In this section I am going to demonstrate how these two excursive topics may be related. Some early references have already been made through the introductory sections, here I will provide further explanation.

The goal of this thesis was to find data mining tools which could facilitate the endurance test result evaluation. Immediately after setting this objective, some important questions had emerged: how to reformulate test result evaluation into data mining problem, how can one create suitable model from the given data. The key to being able to answer these questions was to learn how endurance tests are run and what kind of data is available.

As mentioned in subsubsection 2.2.1, during each test run a certain number of parameters have been measured every fifteen minutes and the results have been stored in separate XML files. Luckily, the software testers had already labelled each previous test run as okay (ok) or not okay (nok), so it was obvious that based on those labels I would be able to annotate the XML files containing the measurement results. In this way, the domain knowledge of human testers also would be preserved, and hopefully it would contribute to achieving better results.

Reasonably, the initial data consisted of the XML files, in which the test run measurements were stored. The annotation of the data was effortless on account of the labelled test runs, but unfortunately I did not manage to categorise all of the gathered data due to the fact that test run labels sent through emails were lost or due to lack of information we were not able to identify the test run to which the label should have been matched. Related to the used up data, I provide further details in section 6, here I continue with the formulation of adequate data mining task, which can be solved using the initial data as input.

So, the question remained: how can we use the labelled data and what data mining task should be formulated.

Given the input data I wanted to predict the label of new test runs based on the results of the previous ones. This goal easily could be associated with one of the core data mining tasks: the predictive modelling. Since the labels of the test runs could be only ok or nok, I was going to face a classification data mining problem.

As revealed in subsubsection 3.2.1, there are already plenty of algorithms available, which, after trained on the input data, are able to deal with generalisation of
the information, and label new inputs according to that. Applying some of these algorithms on our pre-processed data could be the first reasonable step. Furthermore, after understanding our data and learning how the baseline algorithms work, with subtle modifications of these algorithms even better results could be achieved.

Fortunately, the above mentioned algorithms already have been implemented in various programming languages, applying them on pre-processed data does not require too much effort. Pre-processing, on the other hand, seemed to be much more demanding because of the form of the initial data and the complexity of it. Suitability for both of these steps was one of the priorities when choosing a programming language to work with.

The next section is intended to present the development environment in which all of the work related to the thesis has been done, besides presenting the chosen programming language and some useful libraries of it.
5 Development environment

5.1 Environment

As the tested IMS applications were products of Ericsson company, and the test run results were considered confidential, the whole work related to the thesis has been done at the company, on a laptop with the following characteristics:

- Operating system: 64-bit ubuntu 16.04 LTS
- Processor: Intel® Core™ i7 – 4600M CPU @ 2.90 GHz x 4
- Memory: 15.6 GB, swap space: 15.6 GB.

Favourably, the Jupyter Notebook is a cross-platform client-server application [8], thus runs on Linux operating systems too. It allows editing and running notebook documents via a web browser and enables the users to work with different kernels for the various programming languages. With the IPython kernel [9], [10] for using python programming language, it is frequently used environment for data analysis because of its versatility: it allows for data cleaning and transformation, statistical modelling, machine learning, interactive widgets to manipulate and visualise data in real-time, also ensures possibility to handle big data. Hence, it has been suitable for all of the work that has been required for this thesis.

To easily manage the software and library installations, I used from the beginning the Anaconda distribution, which is the leading open data science platform [11].

5.2 Programming language and libraries

Regarding the programming language, Python has been used during the whole progress, including pre-processing of data, algorithm development and data visualisation too. The reason of choosing this language was simple: Python allows for fast and dynamic way of development with a great variety of data mining and visualisation libraries, thus it is perfectly fit for purpose.

This choice of programming language has proven to be a good one, because learning the language itself and the use of different libraries was simple and it fastened my work a lot.

Concerning the diverse libraries I used, I highlight here only the most important ones, which has proven to be very useful when dealing with big amount of data, data mining, and visualisation. The numpy [12] and pandas [13] libraries provide high-performance, easy-to-use data structures and data analysis tools. The scipy
library is practical when advanced math, signal processing, optimisation, statistics are needed. In the scikit-learn library I found the implementation of every necessary data mining algorithm, but I also have learnt that the Modular toolkit for Data Processing (MDP) library also is a great collection of supervised and unsupervised learning algorithms. For data visualisation I mainly used matplotlib and bokeh, additionally the seaborn library, which instantly made the plots visually more appealing, usually I just imported it with that certain purpose. For visualisation of decision trees I used the pydotplus library.

The Configparser and glob built-in libraries of Python helped a lot in organising the written programme codes and handling data collected from different directories in the file system. For parsing XML files I used the cElementTree module from the xml.etree built-in python library, which simplified the work with XML files. These libraries and the version of them which I used are summarised in Table 1.

Parsing the data from the XML files was only a small step of pre-processing. In order to apply data mining algorithms, the initial data needed to be transformed. In the following section I describe how the data sets have been created from the input data, also I provide a brief characterisation of the data set.

<table>
<thead>
<tr>
<th>OS</th>
<th>Linux 4.4.0 75 generic x86_64 with debian stretch sid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>2.7.13 64bit [GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]</td>
</tr>
<tr>
<td>Anaconda</td>
<td>4.3.17</td>
</tr>
<tr>
<td>Jupyter</td>
<td>1.0.0</td>
</tr>
<tr>
<td>IPython</td>
<td>5.1.0</td>
</tr>
<tr>
<td>numpy</td>
<td>1.12.1</td>
</tr>
<tr>
<td>scipy</td>
<td>0.19.0</td>
</tr>
<tr>
<td>pandas</td>
<td>0.19.2</td>
</tr>
<tr>
<td>matplotlib</td>
<td>2.0.1</td>
</tr>
<tr>
<td>bokeh</td>
<td>0.12.4</td>
</tr>
<tr>
<td>seaborn</td>
<td>0.7.1</td>
</tr>
<tr>
<td>sklearn</td>
<td>0.18.1</td>
</tr>
<tr>
<td>xgboost</td>
<td>0.4</td>
</tr>
<tr>
<td>pydotplus</td>
<td>2.0.2</td>
</tr>
</tbody>
</table>

Table 1: Used software and their version
6 Data sets

As mentioned before, the input data was real-world data: measurements from stability test runs of IMS applications. As generally in case of data mining projects, the pre-processing of the input data required the bulk of the effort.

In order to be able to generalise and also to verify every step of the work, I planned from the beginning to work with data from two differently configured IMS applications, thus I created two separate data sets.

The first step in the work process was data collection and annotation. The measurements from previous test runs have been stored in XML files, for each test run in separate directory, which meant approximately 60 GB of data from two different sources. Collection of the data was already done, I just had to copy it in my own repository.

The annotation of the data was a little bit trickier. The software testers have sent reports, in which they mentioned the test run and they labelled it as ok or nok. I had to match a pack of report emails with the corresponding test run measurements. After assembling a list of test runs and their labels based on the emails, I only had to sort the test run measurements based on it. Unluckily, some of the emails referred to test runs from which I did not have measurements and there were test runs to which I could not find label in emails. Thus, a part of the data remained unlabelled, as can be seen in Table 2. Table 2 summarises the number of test runs from both of the data sources, which I worked with, and similarly the number of XML files from both of the data sources.

From the data of the Table 2 one can easily see that data from the first source was slightly favourable, because the number of both the ok and nok labelled XML files was a little larger and besides, the number of unlabelled XML files was significantly smaller.

The next phase involved parsing the XML files in order to collect the data stored in them. With the use of xml.etree library, this task became very easy, basically

<table>
<thead>
<tr>
<th>Number of test runs</th>
<th>Number of XML files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First source</td>
</tr>
<tr>
<td>OK</td>
<td>66</td>
</tr>
<tr>
<td>NOK</td>
<td>5</td>
</tr>
<tr>
<td>UNLABELLED</td>
<td>72</td>
</tr>
<tr>
<td>ALL</td>
<td>143</td>
</tr>
</tbody>
</table>

Table 2: Number of test runs and number of XML files
I just had to examine the structure of the XML files. In Figure 1 can be seen the structure of such an XML file. The interesting parts were the <mt> tags containing the names of the different measured parameters, the <moid> tags containing ids which described the hardware location of the source, and the <r> tags containing the concrete measured values.

In order to understand the data itself, first of all I made some statistical measurements and histograms which pointed out basic information related to the number of parameters, ids, and the number of different value-types.

Concerning the value-types, I distinguished three categories of the measured values:

- **NaN value** - not a number
- **equal to zero**
- **positive value**.

In the case of the different value-types, the histograms Figure 2, Figure 3, and Figure 4 show how frequently the number of NaN values, zeros, and positive values occur among the number of the different value-types for each parameter.
Figure 2: Histogram - number of NaN values

Figure 3: Histogram - number of zeros
Figure 4: Histogram - number of positive values

Figure 5: Histogram - number of ids for each parameter
From the statistical measurements I also found out that basically there are 1326 parameters measured in the first source and 1664 parameters in the second one. Also, I counted the number of ids, the possible hardware locations, for each parameter, and I realised that usually the number of ids for each parameter is small, only a few cases appeared when it was more the one hundred. In Figure 5, the two histograms show how frequently the number of ids occur among the number of ids for each parameter, in case of both data sources. Instead of separately considering parameters and ids, I decided to pair the parameters and ids, and consider them as features. In this way, the number of features on the two sources were 10131 and 59928, respectively.

An important step was to measure the number of occurrences of each parameter-id pair in each XML file. From this measurement I found out that normally each parameter-id pair occurs exactly once in each file, and it occurs only if it has been measured. Also it turned out that there were 51 XML files in case of the first application, in which each parameter-id pair has occurred twice, but with the same measured values, so in order to avoid the problem of these duplicated values, I kept only the last occurred value of each parameter-id pair simply by overwriting the previous value during the parsing phase.

After collecting all of the parameter-id pairs from both sources, I decided to create a table of data in which the columns would be the parameter-id pairs, and each row would contain the data from a single XML file. Merging data of XML files from the different measurements in a single table meant that I could not apply time series approach. Applying time series approach would have been possible only if I considered separately each test run and its measurements, but the low number of test runs and the difficulty of choosing appropriate similarity measure of time series were against following this method.

So, finally I ended up with two tables of data stored in CSV files from the two different sources. The table of data from the first source contained 10131 columns, and as rows: 5076 samples of data labelled as ok, 799 samples of data labelled as nok and 4365 unlabelled data samples, 10240 rows in total. The table of data from the second source contained 59984 columns, and as rows: 4440 samples of data labelled as ok, 666 samples of data labelled as nok and 13330 unlabelled data samples, 18436 rows in total.

Since the parameter-id pairs in the columns were considered as features, reasonable feature selection definitely was necessary due to large number of parameter-id pairs which create a high dimensional feature space. Firstly, the software tester experts highlighted the one hundred most important parameters that always should be
checked. After selecting parameter-id pairs containing only these parameters resulted in 731 columns which still was a high number. So, I dropped even more columns. First of all, I dropped columns filled only with NaN values and columns in which there was any NaN value, thus I did not have to handle the missing value problem and in the meantime this method reduced the number of features. Secondly, I also dropped columns where the mean of values in the column was equal to 0. This drop was reasonable, because in this case it was certain that negative values were not present, so if the mean of values in the column was equal to 0, it meant that the whole column contained only zeros. This also meant that it did not contain information based on which possibly the ok measurements could be distinguished from the nok ones.

After all of these transformations were performed on the data from both sources, I had two data sets: one containing the relevant data from the first source with 328 parameter-id pairs as features, and the other one containing the relevant data from the second source with 4027 parameter-id pairs as features. The larger number of features in the second data set. Also I mention here: both of the data sets contained all of the data from labelled and unlabelled XML files, so when applying the different models, I considered selecting only the labelled data for supervised models, and in case of unsupervised models I used the unlabelled data only for training the model, in order to be able to evaluate the results.

Concerning the data structure, I used the DataFrame data structure from pandas library [21], which has to_csv and read methods, thus it made easy to load the data saved in CSV files in form of a matrix. In order to preserve the hierarchical structure of the header of my datatables, I used MultiIndex for the columns [22]. In this way the parameter names became the level 0 identifiers and the ids the level 1 identifiers, and this have made possible to select and slice the data in DataFrame based on only the parameters or only ids.

After this pre-processing phase, I began the search for adequate data mining models in order to solve the previously formulated data mining task. In he following section are described the data mining models, which I applied afterwards on both of the created data sets.
7 Applied data mining models

In this section, after laying out the foundations of supervised and unsupervised learning and binary classification, I introduce the different models, which have been applied on both of the two constructed data sets in order to solve the proposed data mining task.

Firstly, a basic classification algorithm is described: the Naive Bayes, which was considered as baseline model. After that the Decision Trees and the ensemble methods are introduced: Random Forest, Adaptive Boosting and Extreme Gradient Boosting, which is the leading-edge method currently. Finally, two state-of-the-art unsupervised methods are presented: the One-Class Support Vector Machine and the Isolation Forest.

In this section from now on I assume that $X$ is the data sample set, and $Y$ is the class-label set, and $F$ is the feature set. Elements of an $X$ data set are the $x_i$, $i = 1, 2, \ldots, m$ samples, elements of the feature set $F$ are $f_j$, $j = 1, 2, \ldots, n$, meaning that the $X$ data set has $m$ samples or records, each with $n$ feature. The elements of the class-label set $Y$ are the individual class-labels, $y_k$, $k = 1, 2, \ldots, l$. In case of a binary classification problem the set of class-labels $Y$ contains only two elements: $y_1$ and $y_2$.

Classification formally is the task of learning a target function $T : X \rightarrow Y$ that maps a sample set $X$ to the set of possible class-labels $Y$. If $T$ gives a good approximation for the labels on the training set, then $T(x_i) \approx y_i$, $i = 1, \ldots, m$. Here, target function $T$ is also called the classification model if $Y$ is discrete valued. The decision boundary, associated with a decision function $T$ is an $N$-dimensional hyper-surface which partitions the points in the underlying vector space in two sets, one for each class. A decision boundary is associated with the classifier $C$ if all points in one partition are classified as elements with class-label $y_1$ and all points in the other as elements with class-label $y_2$. In order to be able to provide a formula for each model, it is assumed that \{y_1, y_2\} = \{-1, +1\}.

7.1 Supervised models

In case of supervised learning models the class label of each training data sample is known, the class-label assigned with data sample $x_i$ is marked with $y_i$ in this section. This type of machine learning is useful when reliable domain knowledge could be preserved with labelling the data and the annotation of data is easy.

In the following subsections I present four supervised methods: one, which is a probabilistic classifier: the Naive Bayes, and three ensemble classifiers, which are
based on decision trees: the Random Forest, the Adaptive Boosting and the Extreme Gradient Boosting. The first two of the mentioned ones and related concepts to them are well described in [3].

7.1.1 Naive Bayes

Bayesian classifiers are one of the most simple classifiers, but often are considered just as baseline models, because more advanced and sophisticated classifiers, such as the boosted classifiers in the majority of cases outperform them.

Naive Bayes (NB) classifier is a simple probabilistic classifier based on applying Bayes’ theorem with strong independence assumptions between the features.

Firstly, in order to use Bayes’ approach, the definition of conditional probability is needed. Conditional probability can be defined as a measure of the probability of an event $B$ given that another event, $A$ has occurred and $P(A) > 0$: $P(B|A) = \frac{P(A,B)}{P(A)}$. Bayes’ theorem expresses the relation between the $P(B|A)$ and $P(A|B)$ conditional probabilities: $P(B|A) = \frac{P(A|B) \cdot P(B)}{P(A)}$. Bayes’ theorem can be used for classification.

If $F$ is the feature set, and $Y$ is the class-label set, both can be considered random variables. The two conditional probabilities in this case are:

- prior probability: $P(F|Y)$ assumption
- posterior probability: $P(Y|F)$ prediction

The goal is to predict $Y$, which means the maximisation of the $P(Y|f_1, f_2, \ldots, f_n)$ posterior probability. For the estimation of this probability the Bayes’ theorem can be applied:

$$P(Y|f_1, f_2, \ldots, f_n) = \frac{P(f_1, f_2, \ldots, f_n, Y) \cdot P(Y)}{P(f_1, f_2, \ldots, f_n)}.$$ 

From this it follows that the maximisation of $P(Y|f_1, f_2, \ldots, f_n)$ conditional probability is equivalent with the maximisation of the $P(f_1, f_2, \ldots, f_n, Y) \cdot P(Y)$ product, since the $P(f_1, f_2, \ldots, f_n)$ is constant if only $Y$ changes. The $P(Y)$ prior probability can be estimated from the training data set by calculating the ratio of the training records for each class. NB classifier can be used to estimate the $P(f_1, f_2, \ldots, f_n | Y)$ conditional probability. If we assume that the $f_1, f_2, \ldots, f_n$ features are conditionally independent given a class-label, which is usually considered the naive assumption, then: $P(f_1, f_2, \ldots, f_n | Y = y_k) = \prod_{j=1}^{n} P(f_j | Y = y_k)$, where the $P(f_j | Y = y_k)$ probabilities can be calculated. Finally, the classification method of a unseen $x'$ sample is
the following one:
\[
y' = \arg \max_{y_k} P(y_k) \cdot \prod_{j=1}^{n} P(f_j | Y = y_k).
\]

In case of binary classification this becomes simple: to choose the class-label \(y'\) only a comparison of two products is needed.

When dealing with continuous data, a typical assumption is that the continuous values associated with each class are distributed according to a Gaussian distribution. Then, the probability distribution of a new sample \(x'\) given a class \(y\), \(p(x'|y)\), can be computed by plugging \(x'\) into the equation of a normal distribution parameterised by the mean and variance of values in the training sample set \(X\) associated with class \(y\) \(\mu_y\) and \(\sigma_y^2\):
\[
p(x'|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp \left( -\frac{(x' - \mu_y)^2}{2\sigma_y^2} \right).
\]
Thus, in case of binary classification using the Gaussian NB model, the final classification method is:
\[
y' = \arg \max_{y_k} \{ p(x'|y_k) | k \in \{1, 2\} \}.
\]

7.1.2 Decision trees and ensemble methods

In case of classification, when we try to decide the class-label of a data sample \(x_i\), we can take a sequence of predefined questions with respect to the features of it, and answer each question until we can decide certainly the class-label of the respective sample. The predefined questions and the possible answers to them can be organised in the form of a Decision Tree (DT), which has a hierarchical structure, with nodes and edges between them. A DT has three types of nodes: a root node, internal nodes and leaves. Each leaf node in a DT can be assigned with a class-label. The non-terminal nodes contain test-conditions of features in order to separate the data samples having different features. Such a DT, constructed with the use of a data sample set \(X\) can be utilised to decide the class-label of a previously unseen data sample, \(x'\).

The number of DTs one can construct with \(n\) features is exponentially large, more precisely \(2^n\). This means that the optimal tree cannot be constructed in a reasonable time interval, but some relatively effective algorithms were developed for finding comparatively accurate DTs. These algorithms usually are based on the greedy algorithm where locally-optimal decisions are made at each node regarding which feature is used for separation. Some well-known algorithms used for DT construction are Hunt’s algorithm, the ID3, the C4.5 and Classification And Regression Tree.
known as CART algorithm. These algorithms use different methods and measures to find the best split at each node of a constructed DT.

The best split of the data set is the split that results in the purest subsets, each subset containing a dominant class. Usually, various metrics can be used in order to find the best split, these metrics are defined by the class distribution before and after splits.

The $p(y_k|t)$ class distribution can be defined as the ratio of data samples which belong to class $y_k$ at node $t$. Based on this can be defined the following impurity measures of a node $t$ in the DT:

- Classification error $(t) = 1 - \max_{y_k}(p(y_k|t))$
- Gini index $(t) = 1 - \sum_{i=0}^{l-1} [p(y_k|t)]^2$
- Entropy $(t) = -\sum_{i=0}^{l-1} p(y_k|t) \cdot \log_2 p(y_k|t)$,

where $l$ is the total number of classes. The ID3 and C4.5 algorithms use the Entropy as impurity measure, and by contrast the CART algorithm uses the Gini index.

In order to determine the goodness of a split, the degree of impurity of the parent node before splitting has to be compared with the degree of impurity of the child nodes after splitting. The larger their difference, the better the split. The gain, $\Delta$, is a criterion that can be used to determine the goodness of a split, as well as $\Delta_{info}$, which uses the entropy as impurity measure. The gain ratio also can be used to measure the goodness of a split.

- Gain : $\Delta = I(\text{parent}) - \sum_{j=1}^{n} \frac{N(v_j)}{N} \cdot I(v_j)$
- Information gain : $\Delta_{info} = 1 - \sum_{k=0}^{l-1} [p(y_k|t)]^2$
- Gain ratio = $\frac{\Delta_{info}}{\text{Entropy}(t)}$.

Here, $I(\cdot)$ is the impurity measure of a given node, $N$ is the total number of samples at the parent node, $n$ is the number of features, and $N(v_j)$ is the number of samples associated with the child node $v_j$. DT induction algorithms often choose a test condition that maximises the gain. Since the impurity measure of a parent node is the same for all of the feature test conditions, maximising the gain is equivalent to minimising the weighted average of impurity measures of the child nodes.
In order to increase the accuracy of a classifier, classifier combinations, also known as ensemble methods can be applied. Ensemble methods create a set of base classifiers from the training data and perform classification by taking majority vote on the individual predictions made by each base classifier or additionally weighting each prediction with the accuracy of the base classifier.

There are two necessary conditions for an ensemble classifier to perform better than a single classifier:

I. the base classifiers should be independent of each other

II. the base classifiers should do better than a classifier that performs random guessing

The following three classifiers: the Random Forest, Adaptive Boosting and Extreme Gradient Boosting by default use a DT classifier as base estimator, the only difference between them is how they boost it. This is presented for each classifier in the following paragraphs.

7.1.2.1 Random Forests

Random Forest (RF) classifiers are meta-estimator that fit a number of DT classifiers on various sub-samples of the data set and use averaging to improve the predictive accuracy and control over-fitting.

RFs usually use the Bootstrap Aggregation (Bagging) technique to create the ensemble classifier. Each base classifier DT is created based on an independent random sub-sample of the original data samples and using only a random number of features. The sub-sample usually has the same number of samples as the original data sample set. The independent random sub-samples can be created with bootstrap, in this case the random sub-samples are created with random sampling with replacement and derive from a continuous uniform distribution. Each base estimator DT created with the use of a random sub-sample grows to the maximum height without pruning.

The final classifier decides based on majority votes: a test sample $x'$ is classified by taking a majority vote on the predictions made by the base classifiers $C_q$, and averaging it.

In case the algorithm has built $N$ base classifiers, the final classifier is the following:

$$y' = C^*(x') = \frac{1}{N} \cdot \sum_{q=1}^{N} C_q(x').$$
7.1.2.2 Adaptive Boosting

Adaptive Boosting (AB) classifiers are meta-estimators that begin by fitting a classifier on the original data sample set and then fit additional copies of the classifier on the same sample set but additionally the samples are weighted and the weights are adjusted at each addition iteration phase such that subsequent classifiers focus more on incorrectly classified samples.

AB can be considered an additive model, as it builds sequentially the base estimators, DTs for instance. It applies the boosting iterative procedure, during which the distribution of data samples are modified. A weight is assigned to each \( x_i \) sample, and after each iteration the weights are modified: the misclassified samples get higher weight.

As first step of the AB classification the first base classifier \( C_0 \) is built. After that each subsequent \( C_q \) classifier is built using the weight update method described below.

The weight update method of AB uses the error ratio and the importance of the \( C_q \) classifier for defining the new weights or the \( C_{q+1} \) classifier.

The error ratio for the \( C_q \) classifier can be defined as:

\[
\varepsilon_q = \frac{1}{m} \cdot \sum_{i=1}^{m} \omega_i \cdot \delta(C_q(x_i) \neq y_i).
\]

The importance of \( C_q \) classifier can be defined as:

\[
\alpha_q = \frac{1}{2} \cdot \ln \left( \frac{1 - \varepsilon_q}{\varepsilon_q} \right).
\]

Using these two, the weight update happens according to the following formula:

\[
\omega_i^{(q+1)} = \frac{\omega_i^{(q)}}{z_q} \cdot \begin{cases} e^{-\alpha_q}, & \text{if } C_q(x_i) = y_i \\ e^{\alpha_q}, & \text{if } C_q(x_i) \neq y_i \end{cases},
\]

where \( z_q \) is a normalisation factor to assure that \( \sum_{i=1}^{m} \omega_i^{(q+1)} = 1 \). Additionally, if the error ratio of classifier \( C_q \) is bigger than \( 0.5 \), meaning that it classifies the samples worse than the random guessing, then the new weights become: \( \omega_i = \frac{1}{m} \) for each \( x_i \) sample.

If the number of created base classifiers is \( N \), then the final AB classifier is:

\[
y^{'} = C^*(x^{'}) = \arg\max_{y_k} \sum_{q=1}^{N} \alpha_q \cdot \delta(C_q(x^{'}) = y_k).
\]

Further, more detailed explanation related to the AB classifiers can be read in [23] and [24].
7.1.2.3 Extreme Gradient Boosting

Boosting can be seen as minimisation of a convex loss function over a convex set of functions. Specifically, the loss being minimised by the AB presented in the previous paragraph is the exponential loss function:

$$
\sum_{i=1}^{m} \phi(i, y, C_q) = \sum_{i=1}^{m} \exp(-y_i C_q(x_i)).
$$

In the gradient descent analogy, the output of a classifier for each training data sample is considered to be a point \((C^*(x_1), \ldots, C^*(x_m))\) in \(m\)-dimensional space, where each axis corresponds to a training sample, each base classifier \(C_q(x)\) corresponds to a vector of fixed orientation and length, and the goal is to reach the target point \((y_1, \ldots, y_m)\) or any region where the value of minimised loss function is less than the value at that point, in the least number of steps.

For a given data sample set \(X\) with \(m\) examples and \(n\) features, a tree ensemble model uses \(N\) additive functions to predict the output:

$$
y' = \phi(x') = \sum_{q=1}^{N} C_q(x'),
$$

where \(C_q \in C\) and \(C\) is the space of CART trees where a function \(T\) represents the structure of each tree that maps an example to the corresponding leaf index. Each \(C_i\) corresponds to an independent tree structure \(T\) and leaf weights \(w\). Unlike DTs, each regression tree contains a continuous score on each of the leaf, to represent score on \(i\)-th leaf \(w_i\) is used. For a given sample \(x'\), the decision rules in the trees, given by \(T\), are used to classify it into the leaves and calculate the final prediction by summing up the score in the corresponding leaves, given by \(w\). To learn the set of functions used in the model, the following regularised objective is minimised:

$$
L(\phi) = \sum_i l(y', y_i) + \sum_q \Omega(C_q),
$$

where \(\Omega(C) = \gamma T + \frac{1}{2} \lambda ||w||^2\). Here \(l\) is a differentiable convex loss function that measures the difference between the prediction \(y'\) and the target \(y_i\). The second term \(\Omega\) penalises the complexity of the model. The additional regularisation term helps to smooth the final learnt weights to avoid over-fitting. When the regularisation parameter is set to zero, the objective falls back to the traditional Gradient Boosting.

More detailed description of the XGB can be read in [25].

7.2 Unsupervised models

As it was presented in section 6, in case of both data sets the number of XML files labelled as not okay was significantly smaller than the the number of okay-labelled
ones, and additionally the number of unlabelled XML files also was considerable. The unsupervised models presented in this subsection, the One-Class Support Vector Machine and the Isolation Forest, offer solutions to detecting anomalous samples by taking advantage of these two characteristics of the available data.

### 7.2.1 One-Class Support Vector Machine

One-Class Support Vector Machine (OCSVM) is used for anomaly detection. Given a set of samples \( X \), it will detect the soft boundary of that set so as to classify new points as belonging to that set or not, or providing an anomaly score of samples by calculating their distance from the soft decision boundary.

If \( w \) and \( b \) denote the parameters of the separating hyperplane \( H \), then the signed distance of any point \( x_i \) to \( H \) is given by:

\[
\frac{1}{||w||} \cdot (w^T \cdot x_i + b).
\]

In order to determine the parameters of \( H \) with the largest geometric margin even when the samples are not linearly separable, the SVM approach is to map the samples into an inner product space where linear separation of them is possible and solve a Lagrange dual optimisation problem using a Kernel function to calculate the inner product of two samples in the new space. In most cases the Radial Basis Function Kernel is used for this purpose:

\[
K(x_i, x_j) = \exp(-\sigma ||x_i - x_j||^2),
\]

where \( \sigma > 0 \).

OCSVM was proposed for estimating the support of a high-dimensional distribution. Given data samples \( x_i \in X, i = 1, \ldots, m \) without any class information, the primal problem of OCSVM is:

\[
\begin{align*}
\min_{w, \xi, \rho} & \quad \frac{1}{2} \cdot w^T \cdot w - \rho + \frac{1}{\nu \cdot m} \cdot \sum_{i=1}^{m} \xi_i, \\
\text{subject to:} & \quad w^T \cdot \phi(x_i) \geq \rho - \xi_i, \; \xi_i \geq 0, \; i = 1, \ldots, m,
\end{align*}
\]

where \( \phi(x_i) \) is the transformation function that maps \( x_i \) into a higher-dimensional space and \( \nu \in (0, 1] \). It is proved that \( \nu \) an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. The Lagrange dual problem of OCSVM is:

\[
\begin{align*}
\min_{\alpha} & \quad \frac{1}{2} \cdot \alpha^T \cdot Q \cdot \alpha, \\
\text{subject to:} & \quad 0 \leq \alpha_i \leq \frac{1}{\nu \cdot m}, \; i = 1, \ldots, m, \; e^T \cdot \alpha = 1,
\end{align*}
\]
where \( e^T = [1, \ldots, 1]^T \) is the vector of all ones, \( Q \) is an \((l \times l)\) positive semi-definite matrix, \( Q_{ij} = y_i \cdot y_j \cdot K(x_i, x_j) \) and \( K(x_i, x_j) = \phi(x_i)^T \cdot \phi(x_j) \). Usually the implementations of OCSVM solve a scaled version of the previous equation:

\[
\begin{align*}
\min_{\alpha} & \frac{1}{2} \cdot \alpha^T \cdot Q \cdot \alpha, \\
\text{subject to:} & \quad 0 \leq \alpha_i \leq 1, \ i = 1, \ldots, m, \ e^T \cdot \alpha = \nu \cdot m.
\end{align*}
\]

The final decision function of OCSVM, which provides the anomaly score \( s \) for a data sample \( x' \), is:

\[
s(x') = \sum_{i=1}^{m} \alpha_i \cdot K(x_i, x') - \rho
\]

More detailed description of OCSVM can be read in [26], [27], [28].

### 7.2.2 Isolation Forest

The Isolation Forest (IF) is a relatively new anomaly detection model, it is described more detailed in [29]. An isolation tree (IT) isolates data samples by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. The random partitioning produces noticeably shorter paths for anomalies. Consequently, when a forest of such random ITs collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies. Since the recursive partitioning of the sample set can be represented by the hierarchical IT structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node in the IT. The path length, averaged over a forest of such random ITs, can be perceived as the measure of abnormality.

ITs are constructed according to the following procedure. Given the sample set of data \( X \) of \( m \) instances from a \( d \)-variate distribution, in order to build an IT, \( X \) is divided recursively by randomly selecting a feature \( f_j \) and also randomly selecting a split value \( s \) between the maximum and minimum values, until either:

(i) the tree reaches a height limit specified previously,

(ii) \( |X| = 1 \) or

(iii) all data in \( X \) have the same values.

An iTree constructed in this way is a proper binary tree, where each node in the tree has exactly zero or two daughter nodes.

The path length \( d(x') \) of a point \( x' \) in an IT is measured by the number of edges \( x' \) traverses in the IT from the root node to an external node.
Since an IT has an equivalent structure to a Binary Search Tree (BST), the estimation of the average path length $d(x)$ for external node terminations is the same as the unsuccessful search in BST. Given $m$ samples, the average path length of unsuccessful search in BST is:

$$\hat{d}(m) = 2 \cdot H(m-1) - 2 \cdot \frac{m-1}{m},$$

where $H(\cdot)$ is the harmonic number and $H(i)$ can be estimated by $\ln(i) + \gamma$, where $\gamma$ is the Euler-Mascheroni constant, $\gamma = \lim_{n \to \infty} (-\ln(n) + \sum_{k=1}^{n} \frac{1}{k})$. As $\hat{d}(m)$ is the average of $d(x')$ given $m$, it can be used to normalise $d(x')$.

Finally the anomaly score $s$ of a sample $x'$ can be calculated using $E(d(x'))$, the average of $d(x')$ from a collection of ITs, as:

$$s(x', m) = 2 \exp \left( - \frac{E(d(x'))}{\hat{d}(m)} \right).$$

In the following section is presented how the above described models have been applied, evaluating the results of supervised and unsupervised models separately.
8 Application of models and evaluation

Each of the models described in the previous section has been tested on both of the data sets presented in section 6 in order to get more accurate and reliable results.

For evaluating the obtained results and measuring the performance of models I used the standard performance metrics: the receiver operating characteristic curve (ROC curve), the area under the ROC curve (AUC), and the confusion matrices, from which I was able to easily detect how many test samples were misclassified, as the number misclassified samples appear in the counter-diagonal of a confusion matrix. Besides, I have also measured the training and evaluation time of the different models using the so-called cell magic command of the Jupyter Notebook: \texttt{%%time}. Some other useful built-in ”magic” commands are presented in [30] and [31].

For those who are not familiar with the mentioned performance metrics, in order to understand the evaluation of models in this section, before proceeding with reading is useful to inquire into [3].

The results and evaluation of the models are presented in the ensuing two separate subsections for supervised and unsupervised models. Each model was trained and tested on the data sets separately. Thus, for each model I provide the parameters with which I applied them, the ROC curves of a single training-testing phase and the confusion matrices of a certain threshold, each of these for both data sets. The threshold based on which the number of true positive, false negative, false positive and true negative samples has been calculated for the confusion matrices was automatically selected by each algorithm, except the OCSVM. I marked the selected threshold on the ROC curves with red square.

8.1 Supervised models

In case of the supervised models only the labelled data could be used. In order to evaluate the models on unseen data samples, I used the \texttt{train_test_split} percentage split method implemented in \texttt{sklearn} to randomly divide each data set into two parts, one for training and one for evaluation. For each model I used the \texttt{seed = 7} to generate pseudo-random percentage split. In this way for the evaluation I reserved 33% of the labelled samples, which meant 1939 sample in case of the first data set and 1685 samples in case of the second data set. Luckily, in case of the supervised models there was no need for parameter optimisation, due to either the simplicity of the respective model, or as a result of the complexity of it which had ensured high performance with the default parameters.
8.1.1 Naive Bayes

As NB classifier I used the Gaussian NB implementation from sklearn library [32], I applied it with the following parameter:

```
sklearn.naive_bayes.GaussianNB (priors=None).
```

The algorithm is based on the strong assumption that the data is generated by Gaussian normal distribution. The only parameter of the algorithm, priors refers to the prior probabilities of the classes, which I did not specify.

In Figure 6 can be seen the ROC curves of the NB models. The ROC curves look as it was expected: the NB performs relatively poor on both of the data sets. It is especially interesting that on the second data set initially performs pretty good, but then suddenly the true positive rate stops increasing.

In Figure 7 can be seen the two confusion matrices belonging to the NB models at the threshold marked on the ROC curves. The first confusion matrix indicates that the model performs quite well, it detects more than the half if samples labelled as nok. The second confusion matrix shows that on the second data set the model misclassifies many ok sample as nok by using the selected threshold.

![ROC curves](image)

Figure 6: NB - ROC curves

(a) First data set  
(b) Second data set
8.1.2 Random Forest

As RF classifier I used the implementation of the algorithm in sklearn [33], with the following parameters:

```python
sklearn.ensemble.RandomForestClassifier(bootstrap = False,
    class_weight = None, criterion = 'gini', max_depth = None,
    max_features = 'auto', max_leaf_nodes = None, min_impurity_split =
    1e-07, min_samples_leaf = 1, min_samples_split = 2,
    min_weight_fraction_leaf = 0.0, n_estimators = 10, n_jobs = -1,
    oob_score = False, random_state = seed, verbose = 0, warm_start =
    False).
```

Here, I changed the default value for three of the parameters. I altered the default value of bootstrap parameter from True to False to control the randomness of the model. The n_jobs parameter indicates the number of jobs to run in parallel for both fit and predict. If it is −1, then the number of jobs is set to the number of cores, so I changed the default value to −1. As random_seed I used seed = 7. The implementation uses the Gini index, criterion = 'gini' as impurity measure by default, and min_impurity_split = 1e-7 as lower threshold level for impurity at a node. I did not change these default parameters, and I used the default value for the rest of the parameters too.

In Figure 8 can be seen the ROC curves of the RF models. For better view of the interesting part of them, I zoomed on the upper left corner of the plots and provide here the combined plots of the obtained figures. It was not surprising that, as the ROC curves already indicate, the RF already performed much better than the baseline NB model on both of the data sets, and predicted the labels of unseen samples very good.

Figure 7: NB - Confusion matrices

(a) First data set

(b) Second data set
In Figure 9 can be seen the two confusion matrices belonging to the RF models. The confusion matrices indicate that, using the threshold marked on the ROC curves, the RF misclassifies only 33 samples in case of the first data set, and only 7 in case of the second data set, totally 40 samples, which is already very low, but the AB and XGB outperform even this satisfying results, as it will be presented.
8.1.3 Adaptive Boosting

As AB classifier I used the implementation available in sklearn [34], with the following parameters:

```python
sklearn.ensemble.AdaBoostClassifier(algorithm = 'SAMME.R',
base_estimator = None, learning_rate = 1.0, n_estimators = 50,
random_state = seed)
```

Here I changed only the default random seed to seed = 7. Concerning the rest of the parameters, I used the default values of them. The SAMME.R refers to the real boosting algorithm, while if SAMME is specified for the algorithm parameter, then the SAMME discrete boosting algorithm is used to construct the model. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations. I did not specify the base_estimator for the model, the DT is used by default. The learning_rate parameter shrinks the contribution of the classifier in each iteration by the value provided for it. The n_estimators refers to the maximum number of estimators at which boosting is terminated.

In Figure 10 can be seen the ROC curves of the AB models. As in case of the RF model, I zoomed on the upper left corner of the ROC curve plots and provide here the combined plots of the obtained figures. The ROC curves already indicate that the AB performs approximately as well as the RF, possibly even better.

In Figure 11 can be seen the two confusion matrices belonging to the AB models. These confusion matrices affirm that, at least if using the automatically selected

![AB ROC curves](image)

(a) First data set  
(b) Second data set

Figure 10: AB - ROC curves
threshold, the AB performs slightly better than the RF, as the number of misclassified samples in case of the first data set is 27, and the number of misclassified samples in case of the second data set is 1, resulting in a total of 28 misclassified samples, which is smaller than in case of the RF.

8.1.4 Extreme Gradient Boosting

In the sklearn currently only the traditional Gradient Boosting algorithm is implemented, so the xgboost package in which the improved XGB algorithm is implemented, had to be installed separately [35]. So, as XGB classifier I used the implementation from the xgboost package with the following parameters:

```python
xgboost.XGBClassifier (base_score = 0.5, colsample_bytree = 1, gamma = 0, learning_rate = 0.1, max_delta_step = 0, max_depth = 2, min_child_weight = 1, missing = None, n_estimators = 100, nthread = -1, objective = 'binary:logistic', seed = seed, silent = True, subsample = 1)
```

Here I changed only the default seed to seed = 7. Regarding the rest of the parameters, I used the default value for each of them. The base_score refers to the initial prediction score of all samples. The colsample_bytree is the subsample ratio of features when constructing each CART. The gamma specifies the minimum loss reduction required to make a further partition on a leaf node of the CART. The learning_rate parameter is used for shrinking the feature weights to make the boosting process more conservative. The max_delta_step parameter can help making the model update step more conservative. The max_depth parameter refers to the maximum depth of a CART, increasing its value would make the model more complex and likely to be overfitting. The min_child_weight parameter controls the
preprun of a CART: if the tree partition step results in a leaf node with the sum of instance weight less than the value of this parameter then the building process will give up further partitioning. The \texttt{n_estimators} parameter refers to the number of constructed CARTs during the training, by default its value is 100. The \texttt{nthread} parameter specifies the number of parallel threads used to run, the default value of it is set to maximum number of threads available. Trough the \texttt{objective} parameter the learning task can be specified. The default value of it, 'binary:logistic' refers to the learning task of logistic regression for binary classification. The \texttt{silent} parameter controls whether running messages are printed or not. The \texttt{subsample} parameter refers to the subsample ratio of the training samples.

In Figure 12 can be seen the ROC curves of the XGB models. As in case of the RF and AB models, I zoomed on the upper left corner of the ROC curve plots and provide here the combined plots of the obtained figures. The ROC curves already indicate that the XGB performs approximately as well as the AB, hopefully even better.

In Figure 13 can be seen the two confusion matrices belonging to the XGB models. The confusion matrices affirm that, at least using the automatically selected threshold, the XGB performs slightly better than the AB on the first data set, as the number of misclassified samples in case of the first data set is 16 only while in case of the AB model it was 27, and the number of misclassified samples in case of the second data set is 1, the same as in case of the AB, resulting in a total of 17 misclassified samples, which is smaller than in case of AB, and the best so far.

![ROC curves](image)

(a) First data set 
(b) Second data set

Figure 12: XGB - ROC curves
8.1.5 Evaluation of supervised models

Given the outstandingly high accuracy of the supervised models, at first it seemed hard to choose the best of them. Based on the number of misclassified test samples, I already presumed that the XGB would prove to be the best performing and most reliable model, but, in order to bear this out, I have used other methods of performance measurement.

In order to get a more reliable measurement of goodness, I randomly divided both data sets into training and test parts using the `train_test_split`, and I repeated this process twenty times with different pseudo-random splits, thus getting twenty different AUC scores for each model separately on the first and second data set. The Table 3 summarises the average and the variance of these AUC scores for each model. The mean of the AUC scores already indicates that the XGB, having the highest mean of AUC scores, proved to be the best performing model. In addition, it is also the most reliable one as the variance of its’ AUC scores is the smallest on both data sets.

<table>
<thead>
<tr>
<th></th>
<th>Mean of AUC</th>
<th>Variance of AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First data set</td>
<td>Second data set</td>
</tr>
<tr>
<td>NB</td>
<td>0.68968</td>
<td>0.65899</td>
</tr>
<tr>
<td>RF</td>
<td>0.98446</td>
<td>0.99903</td>
</tr>
<tr>
<td>AB</td>
<td>0.99264</td>
<td>0.99939</td>
</tr>
<tr>
<td>XGB</td>
<td><strong>0.99267</strong></td>
<td><strong>0.99992</strong></td>
</tr>
</tbody>
</table>

Table 3: Mean and variance of AUC
I visualised the calculated AUC scores on box-and-whisker diagrams. In Figure 15 and Figure 14 can be seen the diagram for each model separately on first and second data set. The diagrams indicate well that the XGB models' AUC scores are the closest to 1, thus it is can be considered the best performing model.

![Figure 14: AUC box-and-whisker diagram -NB](image1.png)

![Figure 15: AUC box-and-whisker diagram - RF, AB and XGB](image2.png)

With the use of the previously calculated means and variances of the AUC scores, I provide here the confidence intervals with 95% reliability for the AUC scores. These confidence intervals indicate that we can be 95% confident that the mean AUC is
Confidence intervals with 95% reliability

<table>
<thead>
<tr>
<th>Model</th>
<th>First data set</th>
<th>Second data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>NB</td>
<td>[0.690527, 0.690745]</td>
<td>[0.658803, 0.658970]</td>
</tr>
<tr>
<td>RF</td>
<td>[0.988461, 0.988470]</td>
<td>[0.999511, 0.999511]</td>
</tr>
<tr>
<td>AB</td>
<td>[0.991944, 0.991951]</td>
<td>[0.998986, 0.998987]</td>
</tr>
<tr>
<td>XGB</td>
<td>[0.996492, 0.996495]</td>
<td>[0.999937, 0.999937]</td>
</tr>
</tbody>
</table>

Table 4: Confidence intervals of AUC

between the lower bound and upper bound of the corresponding confidence interval. Thus, the smaller the range of confidence intervals, and, in this particular case, the bounds of them are closer to 1, the better, more reliable the model is. According to this, the XGB proves to be the best model one more time.

I also measured the runtime of the models. The Table 5 summerises the execution times of the models separately on the first and second data set. I measured separately the model training time and the evaluation time, and additionally calculated the total time consumed as the sum of the training and evaluation time. The runtimes were not so surprising: the most simple model, the NB proved to have be the fastest one, and both the training and evaluation times grew according to the complexity of the model. But this does not mean that the more accurate model should be neglected due to higher runtime. The XGB model has a total runtime under 10 seconds on the first data set, and a runtime under 30 seconds on the second data set. It is fast enough to be applied in real-life situation, and as it proved to be more accurate and reliable on both data sets, it should be the first option to use as supervised model.

<table>
<thead>
<tr>
<th>Model</th>
<th>First data set</th>
<th>Second data set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Training</td>
<td>Evaluation</td>
</tr>
<tr>
<td>NB</td>
<td>63.3 ms</td>
<td>616 ms</td>
</tr>
<tr>
<td>RF</td>
<td>315 ms</td>
<td>912 ms</td>
</tr>
<tr>
<td>AB</td>
<td>2.7 s</td>
<td>1.02 s</td>
</tr>
<tr>
<td>XGB</td>
<td>3.74 s</td>
<td>4.87 s</td>
</tr>
</tbody>
</table>

Table 5: Training, evaluation and total run time of supervised models
8.2 Unsupervised models

In case of unsupervised models it was assumed that class-labels are unavailable in the model training stage, thus I used the labelled data without labels and the unlabelled data altogether to train the models. The labels for the originally labelled data are considered available only in the evaluation stage to compute performance measures, thus in case of the first data set the models were evaluated on 5875 sample and in case of the second data set on 5106 sample.

Both of the applied unsupervised models rely on using the contamination ratio in the training data set. I was able to estimate the contamination ratio in case of both data sets using the information about the labelled data in the following manner. In case of the first data set I knew that 799 samples were labelled as nok out of 5875 sample. A good estimation of contamination proportion in the whole data set is the ratio of the number of nok labelled data and the number of all labelled data. Thus, the estimation of the contamination ratio for the first data set was $\frac{799}{5875} = 0.13$. In the case of the second data set the number of nok labelled samples was 666, and the total number of labelled samples was 5106. Similar estimation in the case of second data set gave as result that the contamination ratio is the same, since $\frac{666}{5106} = 0.13$.

Parameter optimisation for the unsupervised models was necessary as initially their performance was worse than the random guessing. For this process I used the Grid Search method implemented in sklearn [36]. This method performs an exhaustive search over specified parameter values for an estimator, thus finding the best parameters for it.

As both of the unsupervised models provide an anomaly score for each sample instead of class-label, I converted those scores to labels. I used a certain selected threshold of scores, the one based on which the confusion matrices have been calculated. The scores below the threshold have been converted to nok, scores above the threshold have been converted to ok. Using the labels obtained in this way I was able to evaluate the performance of unsupervised models just as if they were supervised ones.

8.2.1 One-Class Support Vector Machine

Before applying the OCSVM, I normalised the data by calculating the z-score of each value in each column, relative to the column mean and standard deviation, using the scipy implementation of z-score normalisation scipy.stats.zscore.

One-class SVM is also implemented in the sklearn library [37], I used it with
the following parameters:

```python
sklearn.svm.OneClassSVM(cache_size = 200, coef0 = 0.0, degree = 3, gamma = 1e-30, kernel = 'rbf', max_iter = -1, nu = 0.13, random_state = seed, shrinking = True, tol = 0.001, verbose = False)
```

Here I changed default value of `gamma`, `nu` and `random_state`. The best performing kernel usually is the Radial Basis Function, `'rbf'`, which is the default kernel for OCSVM, so I did not change it. The `nu` parameter is an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors, I used the initially calculated contamination ratio as estimation for it. The `gamma` parameter, the kernel coefficient if `'rbf'` is used, was the only parameter in case which I had to apply the Grid Search method. As a result, I used the `gamma = 1e-30`, as this value ensured the highest AUC score. In case of the rest of the parameters I used the default values of them.

In Figure 16 can be seen the ROC curves of the OCSVM models. The ROC curves indicate that the OCSVM with the used parameters performs only slightly better than a random guessing on the first data set and interestingly on the second data set at a certain point even starts to perform worse than a random guess.

In Figure 17 can be seen the two confusion matrices belonging to the OCSVM models. I have chosen manually the threshold at which the confusion matrices were calculated. They affirm that OCSVM has a poor performance: it detects wrongly 2429 sample as anomalous or not in case of the first data set, and 2605 sample in case of the second data set, approximately half of the testing samples in both cases.

![ROC curves](image)

(a) First data set  
(b) Second data set

Figure 16: OCSVM - ROC curves
Taking into consideration that in case of the OCSVM the features, based on which a sample is considered anomalous, cannot be extracted, I did not bother with further tuning the parameters of the model. As it turned out, there is a more suitable anomaly detection algorithm for my purposes: the IF.

8.2.2 Isolation Forest

Isolation Forest also has implementation in the *sklearn* library [38], I applied it with the following parameters:

```python
sklearn.ensemble.IsolationForest(bootstrap = False, contamination = 0.13, max_features = 1.0, max_samples = 10, n_estimators = 500, n_jobs = -1, random_state = seed, verbose = 0)
```

Here I have changed the default parameter in case of *contamination*, *n_jobs*, *random_state*, *max_samples* and *n_estimators*. The *contamination* parameter indicates the proportion of anomalies in the data set and it is used when fitting to define the threshold on the decision function. I used here the initial estimation of the contamination proportion. The *n_jobs* parameter indicates the number of jobs to run in parallel for both fit and predict. If it is −1, then the number of jobs is set to the number of cores. As *random_seed* I used *seed = 7*.

In case of the *max_samples* and *n_estimators* I applied the Grid Search method, and instead of the default values I used the *max_samples* = 10 and *n_estimators* = 500 values, meaning that the IF model builds 500 ITs, each one of them with a maximum number of 10 samples.

In Figure 18 can be seen the ROC curves of the IF models. The ROC curves already indicate that the IF performs better than the OCSVM on both data sets. It is especially interesting the ROC curve on the second data set, as at the beginning...
it indicates a model performance similar to the random guess, and then suddenly at a point it begins to get better than that.

In Figure 19 can be seen the two confusion matrices belonging to the IF models. They indicate a more satisfying performance than in case of OCSVM. On the first data set the IF detects wrongly 985 samples as anomalous or not, on the second data set 836 samples, in total 1821 samples, which is significantly fewer than in case of OCSVM, but still would be worthy to understand how the detection works and why the performance of unsupervised models is not as accurate as the performance of the supervised ones. I expound this in the following evaluation of the unsupervised models.

![Figure 18: IF - ROC curves](image)

![Figure 19: IF - Confusion matrices](image)
8.2.3 Evaluation of unsupervised models

As mentioned at the beginning of the section, I evaluated the unsupervised models on the originally labelled samples just as if they were supervised classifications.

The confusion matrices of the models already indicated somehow the goodness of the two model, but for the comparison of the the models based on a more reliable performance measure I used the AUC of the ROC curves presented. The Table 6 summarises the AUC of the two unsupervised models on both of the data sets. It can be easily seen that based on the AUC, the IF outperforms the OCSVM.

I also measured the runtime of the models. The Table 7 summerises the execution times of the models separately on the first and second data set. I measured separately the model training time and the evaluation time, and additionally calculated the total time consumed as the sum of the training and evaluation time. The runtimes were not so surprising: the OCSVM proved to be slower than the IF, especially on the second data set the IF has significantly smaller run times, the total time is below one minute.

<table>
<thead>
<tr>
<th>AUC</th>
<th>First data set</th>
<th>Second data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCSVM</td>
<td>0.59643</td>
<td>0.48841</td>
</tr>
<tr>
<td>IF</td>
<td>0.76848</td>
<td>0.56565</td>
</tr>
</tbody>
</table>

Table 6: AUC of unsupervised models

<table>
<thead>
<tr>
<th>Execution time</th>
<th>First data set</th>
<th>Second data set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Training</td>
<td>Evaluation</td>
</tr>
<tr>
<td>OCSVM</td>
<td>11.1 s</td>
<td>17.6 s</td>
</tr>
<tr>
<td>IF</td>
<td>6.16 s</td>
<td>2.9 s</td>
</tr>
</tbody>
</table>

Table 7: Training, evaluation and total runtime of unsupervised models

As mentioned, in case of the IF it would be worthy to understand how the unsupervised detection works and why the performance of unsupervised model is not as accurate as the performance of the supervised ones. In order to achieve this I began to analyse the samples in case of which the label obtained from the anomaly score did not match with the original class-label.

I found out that in case of the originally nok labelled samples that were labelled as ok using the scores of a model there is a reasonable explanation of the phenomenon. If a test run begins to run as a normal one, but fails at a certain point, then the
measured values until then are normal, significantly different measured values appear only in the ensuing XML files. According to this, it is fairly accepted that not all the nok samples are identified by a model.

In case of the originally ok labelled samples that were labelled as nok using the scores of a model I stumbled upon a more significant and interesting phenomenon. I checked a few feature values of those samples, and realised that some of them are unusually different from the expected values, which indicated that the model labelled them rightly as nok. The software testers also confirmed that indeed, those samples should be considered anomalous in spite of the original ok labelling. This finding meant that the IF is able to detect anomalous samples without relying on the annotation of data and furthermore, probably it detects the anomalous data which remained unobserved by software testers.
9 Decision path

In the interest of finding the parameters which possibly caused faulty behaviour during a test run, I intended to retrieve the decision path of a data sample $x'$ from the tree ensembles of the IF models. Being able to retrieve the decisions based on which a sample was considered nok in this case was equivalent with identifying which parameters have anomalous measured values and should be checked by the testers.

Fortunately, accessing the individual ITs was possible using sklearn.tree module. In order to better understanding of the structure of the individual trees, I used the sklearn.tree.export_graphviz method [39] and the pydotplus library [20] to visualise these trees. In Figure 20 can be seen the visualisation of an IT.

Using the .tree_.feature and .tree_.threshold methods I was able to query the features in the nodes of each individual IT and the selected threshold of each feature. Once the features and thresholds have been stored, I have been able to apply them on samples, in this way being able to define the decision path length in each IT.

Specifically, if a data sample $x'$ is considered anomalous by the IF model, then easily can be found the features based on which it was considered an anomaly. Firstly, one has to find the ITs in which the leaf in which $x'$ lands is the closest to the root meaning that the decision path of $x'$ in those ITs is short. After that the features and thresholds have to be extracted from the nodes of ITs. With the use of them, finally the feature values of $x'$ have to be checked: whether they are lower or upper than the threshold. If A feature value of $x'$ is lower than the threshold for that feature, than that feature is one based on which $x'$ was considered anomaly.

Unfortunately, in case of the best performing supervised model, the XGB, currently, at least in Python, it is not possible to access the individual CARTs as in case of the IT, only the visualisation of them is possible using the plot_tree or to_graphviz methods. The visualisation of an XGB CART can be seen in Figure 21. Thus, in case of XGB models, currently it would be challenging to find a way to extract the decision path of a data sample and provide an implementation for this in Python. In the package of the XGB implemented in R programming language, this is already implemented as far as I know, it may be useful to analyse that before implementing in Python.

If it is necessary to be able to identify which parameters have anomalous measured values, probably it would be more useful to use the AB as supervised instead of XGB, as it performs almost as good, and enables the access of the individual DTs.
Figure 20: IT
Figure 21: XGB - CART
10 Conclusion

Concerning the applied models, from the results can be seen that the supervised XGB model has more than satisfying performance: it labels the the unseen data almost perfectly. However, the unsupervised IF model can also be considered when deciding which model to use in real-life situation, as it is able to detect anomalies in unlabelled data. As annotation is made by human testers, it could contain errors, thus by applying IF it would be possible to avoid building the model on originally mislabelled data. The best solution probably would be to combine the IF with XGB in order to get the most reliable results.

One major improvement from the viewpoint of the test result evaluation could be to store the collected data in relational databases. An attempt was already made to use SQLite databases created with the use of Python’s sqlite3 built-in database application programming interface. Application of the Online Analytical Processing (OLAP) approach also could be considered, which would enable analysis of the multidimensional data interactively from multiple perspectives.

In the feature selection phase, where I kept only the parameters highlighted by the software testers, reasonable feature selection could be performed. By training only with the selected parameters, the models are able to predict the label of class taking into account only those parameters. In order to identify anomalous measured values, the best solution probably would be considering all the rest of the parameters too and performing other methods of dimensionality reduction, like the Singular Value Decomposition (SVD). As many of the features had missing values and were dropped, the first step of this approach could be keeping the columns containing missing values, and replacing the missing values with reasonable ones: with previous or next value in the column for instance. Another approach would be to use models able to handle missing values, the XGB and IF fortunately is capable of this.

Other methods could also be applied in order to detect test runs where some unusual measurements signal the faulty behaviour: for example the Support Vector Data Description (SVDD) could also be tried, it seemed promising at first glimpse.

Regarding the usage of models in real-world situation, each model has either decision_function or predict_proba method, which return scores for each sample. Providing this kind of score instead of the predicted label for a certain XML file would probably be more informative and useful.

The above mentioned options may be part of future work related to the topic, but as the realisation of these suggestions would probably require valuable time, it would be reasonable to select and put into practice only the easily feasible ones resulting in the best improving.
REFERENCES

[1] Software testing. 
URL: https://en.wikipedia.org/wiki/Software_testing.

URL: http://istqbexamcertification.com/what-is-a-software-testing.


[4] Jiawei Han, Jian Pei, and Micheline Kamber. Data mining: concepts and techniques. Elsevier, 2011.


URL: https://pdfs.semanticscholar.org/13a4/e7d48a80413ab49bada9e7c005b0b341600a.pdf.

URL: https://pdfs.semanticscholar.org/272b/0a9267b180f431a10faad77ee7dc3a98eccf.pdf.


URL: https://www.continuum.io/.

[12] *Numpy library.*

URL: http://www.numpy.org/.


URL: http://pandas.pydata.org/index.html.

[14] *Scipy library.*


URL: http://scikit-learn.org/stable/.

[16] *Modular toolkit for Data Processing.*


URL: http://matplotlib.org/.

[18] *Bokeh library.*


[19] *Seaborn library.*

URL: https://seaborn.pydata.org/.


URL: http://pydotplus.readthedocs.io/.

[21] *Pandas DataFrame object.*


[22] *Pandas MultiIndex index object.*


URL: https://cseweb.ucsd.edu/~yfreund/papers/IntroToBoosting.pdf.


  DOI: 10.1162/089976601750264965.
  URL: https://doi.org/10.1162/089976601750264965.

[27] Chih-Chung Chang and Chih-Jen Lin. “LIBSVM: a library for support vector machines”. In: ACM Transactions on Intelligent Systems and Technology (TIST) 2.3 (2011), p. 27.


  URL: https://cs.nju.edu.cn/zhouzh/zhouzh.files/publication/icdm08b.pdf.


[31] Jupyter Notebook tips, tricks and shortcuts.
  URL: https://www.dataquest.io/blog/jupyter-notebook-tips-tricks-shortcuts/.

Random Forest.

Adaptive Boosting.

Extreme Gradient Boosting.

Jupyter Notebook tips, tricks and shortcuts.

One-Class Support Vector Machine.

Isolation Forest.

Export_graphviz module.