Perttu Piippo

AUTOMATIC CAPACITY TEST CASE RESULT ANALYSIS FROM CPU LOADS

Master’s Thesis
Degree Programme in Computer Science and Engineering
November 2018

ABSTRACT

Machine learning and artificial intelligence have become popular buzz words in the science community. The advancements in both methodology and computing capability have paved the way for major improvements in multiple fields, leading to an increased interest in machine learning. Most importantly, many companies have also shown interest in applying machine learning in real life situations.

Automated testing environments is one area where there is both data and application for machine learning available. For example, software and hardware capacity testing can produce large amounts of result data that can become unfeasible to analyze manually. Introducing machine learning into these environments can both help find interesting insights and automate the monitoring of results. This thesis explores introducing machine learning into the capacity testing environment of a major telecommunications company.

The work involved exploration of available testing data and various machine learning algorithms. Clustering, anomaly detection and regression with multiple methods were experimented with using different data sets. Ultimately regression using random forests to estimate base station CPU loads based on LTE L2 test parameters was found to produce most promising results.

Two random forest models were built and optimized with hyperparameter tuning and trained and validated with real test result data. The validation showed that the models were valid and could explain 99% of the variation in CPU loads. The most important features affecting CPU loads were also identified. The predictive models thus created are utilized in an automatic monitoring tool that reports possibly anomalous test results based on measured CPU loads. Additionally, this work acts as a groundwork for further machine learning efforts in the target environment and highlights the need for a better data plan in the company for better utilization of data science methods.

Keywords: Data mining, LTE, Machine learning, Random forest
Koneoppimisesta ja tekoälystä on tullut polttavia puheenaiheita tiedeyhteisön keskuudessa. Sekä menetelmiltä että tietotekniikan kehitys ovat mahdollistanet merkittäväät edistyksaskeleet monella aihealueella. Tämä on johtanut kiinnostuksen merkittävään kasvuun koneoppimista kohtaan. Ehkä kaikkein huomattavinta on yhtöiden kasvava kiinnostus soveltaa koneoppimista käytännön käyttötarkoituksiin.


Avainsanat: Koneoppiminen, LTE, Random forest, Tiedonlouhinta
# TABLE OF CONTENTS

**ABSTRACT**

**TIIVISTELMÄ**

**TABLE OF CONTENTS**

**FOREWORD**

**ABBREVIATIONS**

1. **INTRODUCTION** ................................................................. 9
   1.1. Contribution .................................................................................. 10
   1.2. Scope of Thesis .............................................................................. 11

2. **REVIEW OF LITERATURE ON MACHINE LEARNING** ....................... 12
   2.1. Machine Learning ......................................................................... 12
   2.2. Uncertainty and Decision Making .................................................. 13
   2.3. Classification ................................................................................ 15
      2.3.1. Logistic Regression ................................................................. 15
      2.3.2. k-Nearest Neighbors .............................................................. 16
      2.3.3. Support Vector Machines ......................................................... 17
   2.4. Clustering ..................................................................................... 18
      2.4.1. K-means Clustering ................................................................. 18
      2.4.2. Anomaly Detection ................................................................. 19
   2.5. Regression .................................................................................... 20
      2.5.1. Linear Models ......................................................................... 20
      2.5.2. Nonlinear Models .................................................................... 21
   2.6. Decision Trees .............................................................................. 22
      2.6.1. Classification and Regression Trees (CART) ......................... 22
      2.6.2. Bagging and Boosting .............................................................. 24
      2.6.3. Random Forests ...................................................................... 25
      2.6.4. Isolation Trees ........................................................................ 25
   2.7. Neural Networks .......................................................................... 26
      2.7.1. Structure ................................................................................. 26
      2.7.2. Deep Learning ........................................................................ 27
      2.7.3. Applications ............................................................................ 28
   2.8. Evaluation and Feature Selection .................................................. 29
      2.8.1. Evaluation Methods and Measures ........................................... 29
      2.8.2. Feature Selection and Feature Importance .............................. 32

3. **LONG TERM EVOLUTION AND THE TARGET ENVIRONMENT** ........ 34
   3.1. LTE ............................................................................................. 34
      3.1.1. LTE Architecture .................................................................... 35
      3.1.2. LTE Radio Protocols ............................................................... 36
   3.2. Testing Environment ..................................................................... 40

4. **DATA MINING AND PARAMETER DEFINITION** ................................. 42
   4.1. Defining Available Data ............................................................... 42
      4.1.1. Databases ............................................................................... 43
      4.1.2. Job Artefacts .......................................................................... 43
   4.2. Test Case Parameter Definition .................................................... 44
      4.2.1. Choosing Parameters .............................................................. 44
      4.2.2. Log Parsing ............................................................................ 45

5. **EXPLORATIVE ANALYSIS** .............................................................. 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Python libraries</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Database Data</td>
<td>47</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Test Setup</td>
<td>47</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Results and Discussion</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Job Artefact Data</td>
<td>51</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Test Setup</td>
<td>51</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Results</td>
<td>53</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Discussion of Results</td>
<td>59</td>
</tr>
<tr>
<td>6.</td>
<td>IMPLEMENTATION AND EVALUATION</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Test Parameters and Data Set</td>
<td>65</td>
</tr>
<tr>
<td>6.2</td>
<td>Model Selection</td>
<td>66</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Random Forest Regressor</td>
<td>66</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Hyperparameter Selection</td>
<td>67</td>
</tr>
<tr>
<td>6.3</td>
<td>Model Validation and Importance of Features</td>
<td>72</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Model 1</td>
<td>73</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Model 2</td>
<td>76</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Wrap-up</td>
<td>79</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance Monitoring Tool</td>
<td>80</td>
</tr>
<tr>
<td>7.</td>
<td>DISCUSSION</td>
<td>83</td>
</tr>
<tr>
<td>7.1</td>
<td>Future work</td>
<td>85</td>
</tr>
<tr>
<td>8.</td>
<td>CONCLUSION</td>
<td>87</td>
</tr>
<tr>
<td>9.</td>
<td>REFERENCES</td>
<td>88</td>
</tr>
<tr>
<td>10.</td>
<td>APPENDICES</td>
<td>94</td>
</tr>
</tbody>
</table>
FOREWORD

This thesis was an individual effort, but in the larger context it was part of a bigger effort at the company to implement machine learning in their systems. While it is but a small part in this grand endeavor, I hope that the results of my work prove useful to the company and can help those that continue developing solutions for automatic result analysis in the LTE L2 environment and other branches of the company. This work has been a great experience and has provided me with valuable information about machine learning, capacity testing and the development and maintenance of LTE L2 software. I am sure this knowledge will prove useful in the future.

I would like to thank Nokia Networks for this opportunity to work with their data and systems. Furthermore, I would like to thank my technical supervisor Kirsti Simula and secondary supervisor Jouko Lindfors for providing guidance and helping to get this thesis done with dignity. I would also like to thank my line manager Heli Ketola and the CAPA SCT squad and any other coworkers that have helped during this process for their support. Additionally, I would like to thank my former, now retired, line manager Jorma Taramaa for originally giving me the opportunity to work at Nokia Networks and being supportive and encouraging during the time I worked under his supervision.

From the University of Oulu, I would like to thank my supervisor Satu Tamminen for overseeing my work and offering valuable help during the process. I would also like to thank Juha Röning for acting as the second examiner of my thesis. The University of Oulu and especially the CSE department deserve my gratitude for the amazing years I spent there and for providing me with an excellent education. Outside of both Nokia and the university I would like to thank Vanessa Sjögren for proofreading this thesis and making it much more readable.

Finally, I would like to give thanks to my friends and family, who have provided me support during the making of this thesis and kept me sane when it felt like the sky was falling. Your presence has been an invaluable source of strength.

Oulu, Finland November 26, 2018

Perttu Piippo
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>Number of Features</td>
</tr>
<tr>
<td>n</td>
<td>Number of Samples</td>
</tr>
<tr>
<td>R²</td>
<td>Coefficient of determination</td>
</tr>
<tr>
<td>X</td>
<td>Input Vector</td>
</tr>
<tr>
<td>xₖ</td>
<td>kth Input Parameter</td>
</tr>
<tr>
<td>Y</td>
<td>Output Vector</td>
</tr>
<tr>
<td>Ŷ</td>
<td>Predicted Output Vector</td>
</tr>
<tr>
<td>yₖ</td>
<td>kth Output Target</td>
</tr>
<tr>
<td>3GPP</td>
<td>3rd Generation Partnership Project</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AUC</td>
<td>Area Under Curve</td>
</tr>
<tr>
<td>CART</td>
<td>Classification and Regression Trees</td>
</tr>
<tr>
<td>CI</td>
<td>Continuous Integration</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>C-RAN</td>
<td>Cloud Radio Access Network</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>DRB</td>
<td>Data Radio Bearer</td>
</tr>
<tr>
<td>E-UTRAN</td>
<td>Evolved Universal Terrestrial Radio Access Network</td>
</tr>
<tr>
<td>ENodeB</td>
<td>Evolved Node B</td>
</tr>
<tr>
<td>EPC</td>
<td>Evolved Packet Core</td>
</tr>
<tr>
<td>EPS</td>
<td>Evolved Packet System</td>
</tr>
<tr>
<td>HARQ</td>
<td>Hybrid Automatic Repeat Request</td>
</tr>
<tr>
<td>HSS</td>
<td>Home Subscriber Server</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>k-NN</td>
<td>k-Nearest Neighbours</td>
</tr>
<tr>
<td>L1</td>
<td>Layer 1</td>
</tr>
<tr>
<td>L2</td>
<td>Layer 2</td>
</tr>
<tr>
<td>L3</td>
<td>Layer 3</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MAE</td>
<td>Mean Absolute Error</td>
</tr>
<tr>
<td>MHz</td>
<td>Megahertz</td>
</tr>
<tr>
<td>MIMO</td>
<td>Multiple Input / Multiple Output</td>
</tr>
<tr>
<td>MLE</td>
<td>Maximum Likelihood Estimation</td>
</tr>
<tr>
<td>MLP</td>
<td>Multi-Layered Perceptron</td>
</tr>
<tr>
<td>MME</td>
<td>Mobility Management Entity</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Squared Error</td>
</tr>
<tr>
<td>NAS</td>
<td>Non-Access Stratum</td>
</tr>
<tr>
<td>OFDMA</td>
<td>Orthogonal Frequency Division Multiplexing Access</td>
</tr>
<tr>
<td>OSI-model</td>
<td>Open Systems Interconnection Reference Model</td>
</tr>
<tr>
<td>P-GW</td>
<td>Packet Data Network Gateway</td>
</tr>
<tr>
<td>PCA</td>
<td>Primary Component Analysis</td>
</tr>
<tr>
<td>PCRF</td>
<td>Policy and Charging Function</td>
</tr>
<tr>
<td>PDCP</td>
<td>Packet Data Convergence Protocol</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>PDU</td>
<td>Protocol Data Unit</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical Layer</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality-of-Service</td>
</tr>
<tr>
<td>RAN</td>
<td>Radio Access Network</td>
</tr>
<tr>
<td>RB</td>
<td>Radio Bearer</td>
</tr>
<tr>
<td>RLC</td>
<td>Radio Link Control</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent Neural Network</td>
</tr>
<tr>
<td>ROC</td>
<td>Receiver Operating Characteristic</td>
</tr>
<tr>
<td>ROHC</td>
<td>Robust Header Compression</td>
</tr>
<tr>
<td>RRC</td>
<td>Radio Resource Control</td>
</tr>
<tr>
<td>S-GW</td>
<td>Serving Gateway</td>
</tr>
<tr>
<td>SAE</td>
<td>System Architecture Evolution</td>
</tr>
<tr>
<td>SBS</td>
<td>Sequential Backward Selection</td>
</tr>
<tr>
<td>SC-FDMA</td>
<td>Single Carrier – Frequency Division Multiple Access</td>
</tr>
<tr>
<td>SDU</td>
<td>Service Data Unit</td>
</tr>
<tr>
<td>SFS</td>
<td>Sequential Forward Selection</td>
</tr>
<tr>
<td>SBFS</td>
<td>Sequential Backward Floating Selection</td>
</tr>
<tr>
<td>SFFS</td>
<td>Sequential Forward Floating Selection</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SRB</td>
<td>Signaling Radio Bearer</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Testing</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Machine</td>
</tr>
<tr>
<td>TB</td>
<td>Transport Block</td>
</tr>
<tr>
<td>TTI</td>
<td>Transmission Time Interval</td>
</tr>
<tr>
<td>UE</td>
<td>User Equipment</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>UTRAN</td>
<td>Universal Terrestrial Radio Access Network</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice over Internet Protocol</td>
</tr>
<tr>
<td>WCDMA</td>
<td>Wideband Code Division Multiple Access</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Machine learning can be found in many, sometimes surprising, places in today’s world, from smartphones to web browsers. Machine learning runs in the background of search engines, social media content filtering, online store recommendations and many other applications that many never think about. While machine learning and its close counterpart Artificial Intelligence (AI) are still far from the fantastic visions of science fiction, it already has its uses, and properly applied can help in tasks too cumbersome for humans.

While machine learning has been a topic of study since computing was invented, it has recently risen in popularity among the scientific community. Advances in computing capabilities have allowed faster and more complex methods to become plausible, while previously they were limited by hardware and regarded more as curiosities. Data, the building block of any machine learning solution, has been gathering over the years, both freely available on the internet and privately available within companies. More advanced methods have been developed, and the current popularity of machine learning research means that more are likely to come.

This work is part of a growing trend in companies of utilising machine learning on real problems. Advances in the field have awoken companies to the possibilities of machine learning, and many have recognised the advantages that implementing intelligent systems can provide. Many have also invested in research on the subject to stay on top of their competition. This is important for the development of the field, as bigger companies like Google and Nokia have both the resources and the data to implement top of the line machine learning solutions. Risto Saalasmaa, the Chairman of the Board of Directors at Nokia, has been vocal about the importance of machine learning and made calls to action within the company [1]. Google has established themselves as an authority in the machine learning field, with their DeepMind researching applications of artificial intelligence and machine learning in a multitude of fields such as the optimization of their own systems and healthcare. Their perhaps most known achievement is AlphaGo, an AI that in 2015 beat a high ranked player in GO, a complex board game that was previously thought too difficult for traditional artificial intelligence methods [2].

One field where machine learning can find practical use is software and component testing. This testing is generally performed to monitor the performance and validity of the artefact being tested. In integration testing two or more individually working artefacts are combined and tested to see whether they work together as planned. Testing environments can also be incorporated with automated testing frameworks, where tests are run with each new change to the code base, both to validate the change and to measure its effect on the artefact under testing.

Testing like this can easily produce large amounts of results. If there are a lot of individual tests and they are run regularly, the amount of data quickly becomes too much for people to manually go through. This can easily lead to less obvious problems and performance issues slipping through the cracks and causing problems later. This is where machine learning can prove useful, as it can be harnessed to automate the process of result monitoring. Where the amount of data is overwhelming for a person, for a learning machine it is not a problem, as machine learning is built upon data.
1.1. Contribution

This thesis is centred around applying machine learning in a testing environment for automatic performance monitoring. The environment belongs to a major telecommunications company and is focused on testing Long Term Evolution (LTE) compliant hardware and software. There is a lot of data available in this environment, which is part of the motivation for this work. This work studies estimating performance in the testing environment with the help of machine learning based on the results that are produced by the various tests runs in the environment. These findings are then meant to be used for automatic performance monitoring in the system based on Central Processing Unit (CPU) loads. In practice, two random forest regression models are presented, capable of estimating CPU loads from the test parameters.

It seems that no similar research has been done on the matter outside of the company. At the very least, any such research is not publicly available. As estimating performance from context specific parameters is most likely done inside companies as a part of product development, the research most likely also stays within the companies. Existing available CPU load estimation research seems to be more resource based and to focus on common factors affecting CPU loads. However, CPU loads and what affects them in similar context has been studied within the company previously. Master’s theses by Kananen [59] and Holappa [60] highlight efforts to estimate CPU loads and what affects them. Especially Kananen’s work is very similar to this work, as it also provides tools to estimate CPU loads from test parameters. The main difference is the methodology and the nature of the tool. Both Kananen and Holappa used manual analysis and provided static tools for estimation, while this work appears to be the first to apply machine learning into this context and introduces a dynamic tool for automated estimation. The number of samples used is also significantly larger in this work.

This work also establishes a groundwork for future machine learning and data science efforts in the target testing environment. In addition to CPU load estimation, the thesis also showcases other possible machine learning and data science applications in the target environment. At the time of this writing there are several other projects starting that rely at least partially on the findings of this thesis. While machine learning is already being studied and applied at other parts of the company, this is the first effort for capacity testing. This work was an individual effort separate from the other machine learning research at the company.

Lastly, this work also attempts to highlight the importance of data planning in systems aiming to implement machine learning. Just having data is not usually enough, there must be enough data and the data must be meaningful. Just as importantly the data should be accessible and preferably in an already usable format. In many real-life situations the majority of the work and effort in machine learning projects goes to locating, identifying, extracting and transforming data, because the system in question does not support the effective handling of the data.
1.2. Scope of Thesis

The scope and complexity of the testing environment currently in use at the company allows and likely warrants a thorough and extensive study into the possibilities of applying machine learning. For this work, focus was put on estimating CPU loads through regression based on the Layer 2 (L2) test parameters and finding the parameters that most affect those loads. This also acts as the main research question for this thesis.

This thesis is structured as follows: chapter 2 presents a literature review into machine learning, highlighting major methods and their achievements. Chapter 3 introduces the company, the LTE architecture and the testing environment. Chapter 4 goes through the process of defining and extracting the data available and choosing the parameters and the data to use in this work. Chapter 5 presents the explorative analysis done to determine the data representation and the choice of machine learning algorithm. The actual models are built and evaluated in chapter 6, and the software tool where the models are utilized is also introduced. Discussion of the findings is presented in chapter 7, and the thesis is concluded in chapter 8.
2. REVIEW OF LITERATURE ON MACHINE LEARNING

Machine learning provides tools for learning from large amounts of data and making decisions based on it. The original goal of this work was to research applying machine learning in an automated testing environment in order to automate and improve performance monitoring in the system. This kind of automated environment provides clear applications for machine learning. The tests provide plenty of data for learning through test results, especially considering the large number of tests and the frequency in which they are executed. The clearly structured, repeating workflow of tests run using test parameters and outputting various results creates an opportunity for using supervised learning to explore their relationship. The workflow also allows inserting both learning and ready machine learning models, automating the process of learning.

While machine learning can be applied to many problems and is potentially a powerful resource, it has its limitations and problems. The recent popularity of both machine learning and artificial intelligence has also introduced misconceptions about machine learning and its capabilities. Many people overestimate what can be done with machine learning. To be able to make educated decisions and understand the results of machine learning a comprehension of the current state of machine learning and its capabilities is required. This literature review is an attempt to provide a sufficient understanding of what machine learning is and what can be done with it.

Common machine learning concepts, methodologies and goals are explored below, with examples of how they can be applied in real life. The focus is on supervised and unsupervised methods, as reinforcement learning is not relevant for this work. This is not an exhaustive list, as the field of machine learning has seen many methods and algorithms over the years. It should also be noted that most machine learning methods can be used for multiple purposes, even if they are assigned to a certain problem here.

2.1. Machine Learning

Machine learning is a field of science and engineering that focuses on giving machines the ability to learn and make decisions based on what they have learned. These machines are exclusively computers, due to which machine learning also falls under the umbrella of computer science. Artificial intelligence is often confusedly thought of as the same as machine learning, but this is incorrect. AI is a more complete imitation of human intelligence, where machine learning provides tools for the AI to achieve its goals [3]. Machine learning is also heavily related to pattern recognition, as both deal with finding and explaining patterns within data. Statistics are also important for machine learning in building and validating models.

There are generally three main types of machine learning. Supervised learning refers to cases where the learning happens using data containing both the input vectors (vectors containing features and their values) and the target vectors (vectors containing the known target values or categories). The data is used to teach the machine the inherent relationships between features that lead to the target values or categories. The input features are also referred to as the independent variables, and the target as the dependent variable. The learnt model can then be used on unknown samples to
categorize them or to estimate the target value [4 p. 2-4, 5 p. 9-11]. Unsupervised learning on the other hand has no target vector. It is more about exploration of the data and the input vector is used to find inherent patterns, such as clusters or distributions, in the data [4 p. 2-4, 5 p. 485-486]. Unsupervised learning is a more desirable form of learning, as it requires no target values and can lead to new discoveries within the data. The downside of unsupervised learning is that the lack of a clear target makes it more difficult to do in practice and the results are more ambiguous and challenging to interpret. Supervised and unsupervised learning can be combined into semi-supervised learning, where target values are known for a portion of the training samples in order to guide the analysis of the other samples.

Reinforcement learning differs from the previous two in that it has no clear input vector or output vector. Instead the machine, or agent as it is usually called, learns from experience. The agent is put within an environment and has a goal or goals to achieve. The agent can perform actions that affect the environment or the agent within the environment. With help of a value function that determines if a set action has a positive or negative effect with the goal in mind, the agent learns through trial-and-error to achieve its goal. An interesting aspect of reinforcement learning is its dilemma between exploration and exploitation, which does not exist in supervised or unsupervised learning. The agent needs to exploit actions previously learnt to be good, but it also needs to explore new actions to learn new and better actions [6 p. 1-11].

There exists many different methods for machine learning, which build different models for different purposes¹. However, data is at the center of all machine learning, as it is the basis on which all methods and models build. Without data there is no learning, and any application built without having it learn from data is simply a program executing a set of instructions set by a human.

### 2.2. Uncertainty and Decision Making

Probability theorem is essential to machine learning and pattern recognition to account for the uncertainty that is present in most real-life situations. Probability densities and distributions like the Gaussian distribution are essential building blocks for pattern recognition methods and models that must deal with uncertainty [4 p. 17-19, 24-28, 7 p. 51-69]. Another very important aspect of probability theorem is the Bayesian viewpoint, although it is not the only one. It can be used to quantify uncertainty and to use prior knowledge or assumptions with actual observations (data) to acquire posterior probability, i.e. the probability of some event happening based on the combination of prior knowledge and observations relevant to said event. The Bayes

Theorem can be expressed as a formula, which popularly takes the form shown in equation (1), where \( A \) and \( B \) are separate events. \( P(B) \) represents the prior probability distribution for \( B \), i.e. the likelihood of event \( B \) happening. \( p(A) \) represents the same for \( A \), conditional probability \( p(A|B) \) represents the likelihood of \( A \) happening if \( B \) is true and \( p(B|A) \) vice versa. \( P(B|A) \) is also called the likelihood function. The equation can also be presented in grammar form (2). The use of proportional to sign \( \propto \) instead of an equal to sign tells that the posterior probability is an estimate based on the likelihood and prior, not the ground truth [4 p. 22-24].

\[
p(B|A) = \frac{p(B|A)p(B)}{p(A)} \quad (1)
\]

\[
posterior \propto likelihood \times prior \quad (2)
\]

While probability theory provides ways to deal with uncertainty and can provide information on the probabilistic description of the situation, real life situations generally demand some concrete prediction or decision based on the probabilities. This is where decision theory comes in, as it can be used in combination with probability theory to make optimal decisions. Decision theory can be used to define rules that determine how to handle the result of inference, i.e. the probability of the target with given observation [4 p. 38-39].

Rules depend on the situation, what is wanted of the model and generally require some expert knowledge to set. Loss, also called cost, is a term used to describe the difference between the estimated and the true value and means different things in different situations. Loss function is a function that calculates this cost. In applications, you generally want to minimize the loss function to achieve the best results, but most real-life situations impose some form of specific rules. For example, in medical applications you might want to minimize the misclassification of sick people as healthy, even if that meant misclassifying more healthy people as sick. In some applications you might also want a certain level of reliability, rejecting observations that cannot be classified with a certain degree of trust or that provide values too far off with regression [4 p. 38-48].

Minimizing the loss function generally requires a specific procedure for estimating the parameters that achieve the smallest error. This is also called fitting, as in fitting the model to the data [4 p. 5-6]. For example, gradient descent is a procedure where initial parameters are chosen randomly, and the error is calculated based on these parameters. The parameters are changed slightly and a gradient, i.e. the rate of change in the error, is calculated. Weights are adjusted to the direction where the gradient is steepest, i.e. the error decreases the most. Weights are adjusted this way until there is no meaningful change the in error any more. Stochastic or sequential gradient descent is a variation of this, where the weights are adjusted after calculating a gradient for each observation [4 p, 144, 240-241, 5 p. 131, 395-397]. Maximum likelihood estimation (MLE) is another fitting method for probabilistic models, where the goal is to find the parameters that maximize the likelihood function. This means that the
selected parameters maximize the likelihood that the model describes the data [4 p. 22-24, 140-143, 5 p. 265-267]. Many other methods for fitting models exist, depending on the model used.

2.3. Classification

Classification is one of the most common machine learning problems. Classification answers the problem of assigning an observation of an unknown class into a class based on previous data. A model that can classify samples is called a classifier. Models can output the assigned class or they can give a probability distribution of what the class could be, which is then interpreted using decision rules. Classification is supervised learning, as the classes of the training observations are required for learning [4. p. 179, 7 p. 98]

2.3.1. Logistic Regression

Despite having “regression” as part of its name, logistic regression is a simple and popular algorithm for a probabilistic binary classification task. It answers the problem of predicting the probability for a dependent variable that has two possible outcomes, where the outcome is dependent on one or more independent variables. A linear regression model alone cannot answer this problem adequately, as it gives values beyond the probability range of zero to one. Instead, logistic regression produces a model that is a logistic, s-shaped curve with values between zero and one that represents the likelihood of a sample having a certain outcome based on the independent variables. A linear function of the predictors is still present, but the curve is calculated using a natural logarithm of the odds (3) called the log-odds or the logit function of the target variable. The odds are calculated using the sigmoid function \( s(t) \) (4), where the \( t \) is the linear function of the predictor \( x \) (5). Linear functions are discussed further in the chapter on Linear Models. In the linear function (5), \( w_0 \) is a constant that in logistic regression moves the curve left and right and \( w_1 \) is the coefficient of predictor \( x \) that defines the steepness of the curve. The coefficients are learned from the training data, and maximum likelihood estimation is generally used with logistic regression [4 p. 196-206, 5 p. 119-122].

\[
\ln \frac{s(t)}{1-s(t)} = t \quad (3)
\]

\[
s(t) = \frac{1}{1+e^{-t}} \quad (4)
\]

\[
t = w_0 + w_1 x \quad (5)
\]
Logistic regression can handle more than one predictor and can deal with both continuous and categorical values. Adding more predictors increases the number of coefficients to learn. Logistic regression can also be extended to deal with multiclass problems, but this requires defining multiple linear functions and a number of coefficients balloons [4 p. 209-210, 5 p. 119-122]. Binary classification is the main use of logistic regression, as it is excellent for cases where we want to classify samples into two categories. Therefore, logistic regression sees frequent use in the medical field where such two-class cases are common. For example, estimating whether a patient survives or whether he has some medical condition or not based on medical and personal data are real-life applications of logistic regression [5 p. 122-124].

2.3.2. \textit{k-Nearest Neighbors}

One simple and popular classification method is called \textit{k-Nearest Neighbors} (\textit{k-NN}). It can also be used for regression, but classification is its most popular application. In \textit{k-NN} classification is done by comparing new observations to their \textit{k} nearest neighbors. The comparison measure is the distance between observations. Euclidean distance is a popular choice for the distance measure, but others such as Mahalanobis distance can be used. For \textit{k}=1, the class of the new observations is the class of the nearest neighbor. When \textit{k} is larger than one, the most common class among the \textit{k} nearest neighbors is chosen for the observation. This can be expanded by adding additional weights to instances based on their distance. In \textit{k-NN}, a specific model of the feature space is not built as is the case in many other methods. Instead the whole data set is one. When a new instance is classified, it is simply compared to the other instances [8 p. 230-234].

While \textit{k-NN} is attractive due to its simplicity and approachability, it has some severe drawbacks. The curse of dimensionality affects nearest-neighbor methods heavily. Because \textit{k-NN} uses all features of an observation for measuring distances, a large number of features can lead to misleading distance measures. This is especially the case if there are only a few features relevant to the classification and many irrelevant features. The irrelevant features can easily dominate the distance between observations. This can be alleviated with weights or removal of irrelevant features, but this requires separate feature selection procedures [8 p. 234-235].

The distance measure itself can be a problem, as choosing the right measure can make or break the classifier. Euclidean distance is commonly used, but it loses its validity with higher dimensions and when the features are uneven. Other distance measures include, but are not limited to, Manhattan, Mahalanobis and Minkowski distances. Another problem comes with the choice for the value of \textit{k}. The number of neighbors considered for classification can affect the outcome [8 p. 234-235].

One last problem with \textit{k-NN} is its computational heaviness. Because it uses the whole dataset as a model, it must load all data into memory and calculate the distance from the new instance to known instances every time it classifies a new observation. This can be eased by data structures such as trees, but bigger data sets are generally a problem [8 p. 236].
Despite its drawbacks $k$-NN finds use in the industry. It requires relatively little knowledge of machine learning and is easy to set up, which makes it compelling for applications where machine learning is only a small part of a bigger process. When its drawbacks are not an issue, it is a well performing algorithm. Machine vision makes use of $k$-NN, as there it can be used for feature matching and object classification. One example is a system that detects whether motorists are wearing helmets, where $k$-NN is utilized for both classifying objects as motorcycles and classifying riders’ heads as wearing a helmet or not [9].

2.3.3. Support Vector Machines

Support Vector Machines (SVM) are one of the most popular and common machine learning methods. They are applicable to both classification and regression problems but tend to see more use in classification. For classification, SVM essentially creates a hyperplane in the feature space that separates the different classes from each other. The distance between the closest data points to the hyperplane is called the margin, and the optimal hyperplane maximizes this margin. In a perfect situation where the classes are completely separable with a hyperplane, the margin is absolute, with no data points falling within it [5 p. 419-422, 10].

As classes tend to overlap in real life, a perfect split with a hyperplane cannot be achieved and the margin has to be relaxed to allow some data points to violate the constraint. Coefficients are introduced to each dimension, which define how much the margin is allowed to give in for each dimension. An additional global parameter, popularly called C, is also defined, which controls the total amount of allowed violations of the margin in the whole feature space. Only the closest data points that lie within the margin are relevant for defining the hyperplane, and they are called support vectors as they support the definition of the hyperplane. C influences the number of support vectors, as it determines how many violations of the margin are allowed. The hyperplane and the coefficients are learned from the data using an optimization method, such as stochastic gradient descent [5 p. 419-422, 10].

The previous description introduces a support vector classifier, but a proper support vector machine is implemented using kernels. Kernels can make computations in high dimension feature spaces much more manageable. Kernels are functions that define the similarity or distance measure between different observations and that can be used to compare new observations against support vectors to make predictions. Kernels take advantage of the fact that the inner products of any two observation vectors can be used instead of the observations themselves. This means that both training and testing functions are only dependent on the data through the kernels. In linear SVM the linear kernel uses the dot product of observations, and for a new observation this is calculated between the new observation and each support vector. More complex hyperplanes can be created using more complex kernels, such as a polynomial kernel for curved lines and a radial kernel for creating decision regions. The general theory here is for a binary support vector classifier, but multiclass SVMs are possible by combining together binary classifiers, where each is a one-versus-rest classifier for each class [5 p. 423-438, 10].
While effective, support vector machines have some caveats. The choice of kernel is probably the most important aspect when building a SVM, but the choice is not apparent in all cases. Another problem comes with training and testing size and speed, as SVM methods can be very computationally expensive with large datasets. Discrete features also require extra care with a SVM [10]. Still, a SVM is a well-established machine learning method and sees use in areas including image processing and recognition, text categorization and the prediction of natural occurrences, such as wind speed [10, 11].

2.4. Clustering

Clustering, also called cluster analysis, is an unsupervised learning problem, where a set of observations are assigned to separate groups based on some similarity criterion. Unlike classification, the categories are not known beforehand, and the clustering method must decide on these clusters based on the training data and assign the samples to them. Clustering is a useful data exploration technique that can be used for discovering groups of observations in the data and finding similarities in observations that might not be apparent with a manual analysis. An established clustering model can also be used to assign new samples into clusters. While clustering is simple in principle, the fact that a clustering method that is over 60 years old and very simple, is still widely used shows that clustering is not an easy problem to tackle [12, 13].

2.4.1. K-means Clustering

Probably the best known and most popular method for finding clusters in data is called K-means clustering. It works by randomly choosing k center points in the feature space of the task at hand, and then calculating the distance between each sample point and each center point. Each point is assigned to the cluster with the closest center point. New cluster centers are then calculated by taking the feature vector of every sample belonging to the cluster and calculating the mean. Each sample is then reclassified based on new cluster centers. These steps are repeated until a set number of iterations is reached or the clusters become stable with little change between iterations. The objective is to minimize the sum of squared errors over all clusters. [4 p. 424-427, 12, 13].

There are a couple of downsides to K-means that reduce its usefulness. First, the choice for the number of clusters is non-trivial. For two or three-dimensional data the number of clusters could be estimated through visualization, but this is not the case for higher dimensional data. One option is to run the algorithm with multiple values of k and decide which one gives the most meaningful result, but even then, a domain expert is needed to choose a meaningful clustering. Another option is to use inertia as an indicator of the optimal number of clusters. Inertia is the within cluster sum of the squares value that the algorithm attempts to minimize. By visualizing the development of inertia with a different number of clusters, one can pinpoint the value of k where inertia starts plateauing, i.e. more clusters bring diminishing improvements.
The random initialization of cluster centers also makes the method inconsistent, as different runs can produce different results due to convergence to local minima. Standard $k$-means also does not work with categorical features, as Euclidean distance is not meaningful in a discrete sample space. [12, 13].

Standardization is also required before using $k$-means clustering. If the variables are of different units or have different variances, not standardizing could skew the results and put more weight on variables with smaller variances. Because $k$-means is isotropic in all directions, the clusters it produces tend to be round, which does not fit well with uneven variables [14]. This isotropic nature of $k$-means also leads it to generally fail in cases where the effective clusters in the data are not circular, or the mean values of the clusters are very similar [12, 13].

Due to its simple nature, easy and efficient implementation and good performance when its downsides are not relevant, $K$-means sees a lot of use as an exemplary algorithm in education. It also sees use in applications despite its flaws. For example, it can be used to cluster behaviour and concepts in the internet [15]. $K$-means and its derivative $k$-medoids have also been used to identify performance degrading code updates in Ericsson’s continuous integration environment based on memory usage and CPU loads in baseband processing units [16]. Image segmentation and compression also make use of $k$-means [4 p. 428-430].

2.4.2. Anomaly Detection

Anomaly detection is a data mining problem that focuses on detecting instances or sequences in data that somehow do not fit the norm of the data and raise suspicion. Machine learning can be used for anomaly detection and it can be considered a subclass of clustering, as it essentially divides observations into two clusters, normal and anomalous. Anomaly detection methods are not strictly clustering, however, as most machine learning methods can be used to detect anomalies [7 p. 100, 17].

Anomaly detection can be supervised, unsupervised or semi-supervised based on the amount of and the nature of the data. The nature of the input data and the availability of labels for samples determines which models and methods are suitable. There are three types of anomalies that can be detected using anomaly detection methods. Single samples that differ from the norm enough to be considered anomalous are called point anomalies, which are the most common and generally easiest to detect. Contextual anomalies are samples that can be normal or anomalous based on the context they appear in. For example, in time series data, a sample might be normal at some point in time but become anomalous if it appears at some other time. Lastly, collective anomalies are data points that alone are not anomalous, but when they appear together become anomalous. A disruption of a pattern in signal data is an example of this [17].

Multiple methods for anomaly detection exist. Classification methods can be used in cases where the data can be used to train a classifier to classify normal and anomalous samples, but are not useful when the data is not labeled for training. Clustering can also be used to detect anomalies, as difficult to cluster samples can be considered anomalous. In particular, $k$-NN based methods work for unsupervised data
and can be used in situations where normal instances are densely together and anomalous instances are farther away. They suffer from computational load with complex data and require correct choice of distance measure to work. Statistical methods, such as regression or histograms, can also be used in situations where normal data can be modeled using a statistical model. Anomalous samples are likely to have a low probability of being explained by the model [17]. Anomaly detection methods are not limited to those presented here.

One popular practical application for anomaly detection is intrusion detection in computer systems. Anomaly detection has seen extensive use on this field, as abnormal behavior can be detected from call traces or network traffic. For example, a probabilistic distribution of system call sequences can be built based on data, and then used to see sequences which don’t fall within it and are thus likely to be anomalous [18]. SVMs have also been used to detect unwanted behavior in payloads of computer traffic [19]. Another popular field for anomaly detection is fraud detection for credit card transactions, insurance claims and other similar domains, where abnormal behavior can imply fraud. Other applications include the medical field, industrial fault detection and sensor data. Anomaly detection in general is potentially useful in any context where anomalous instances can be distinguished from normal instances using data [17].

2.5. Regression

Regression is one of the main categories of machine learning. It has the goal of predicting values of one or more target variables based on a set of input variables. If we have a data set of n observations with a vector \( X \) of values for the input parameters and a vector \( Y \) of corresponding target variable values, a regression model can be trained to predict the values for target variables for a new observation of input variables. A model that can be used for regression is a called a regressor [4 p. 137-138, 7 p. 99]. As most popular methods generally used for regression, such neural networks and random forests, are mentioned under different sections in this literature review, only linear and non-linear models are mentioned in this section.

2.5.1. Linear Models

Linear regression models are linear functions of adjustable parameters, which can be used for regression. Regardless of the form of these models, they aim to model the predictive distribution of the data to obtain a prediction for target variable \( y \) for a new observation \( x \) using \( k \) features that minimizes the chosen loss function. The simplest form of this class of models is called multiple linear regression (6), and it is a linear combination of the input variables. The constant \( w_0 \) is a fixed offset of data, and the other weights define how much each feature affects the outcome. The error term or noise \( e \) quantifies the other factors affecting the dependent variable, which are not included in the input variables \( X \). [4 p. 137-139, 7 p. 105-108, 20 p. 19-21, 47-49]. While simple and easy to interpret, it is the linear function of both parameters \( W \) and input variables \( X \), which limits the model.
\[ y(X, W) = w_0 + w_1x_1 + \cdots + w_kx_k + \epsilon \quad (6) \]

A more flexible class of these models replaces the input variables with nonlinear functions of the input variables. These functions are called basis functions, and can be Gaussian, Fourier based or sigmoidal, for example. This makes them nonlinear functions of the input variables \( X \), while remaining linear models of parameters \( W \). Maintaining the linearity of parameters makes analysis of the models simple [4 p. 138-139].

Least squares method is a common and simple approach for finding the weights, by calculating which weights minimize the sum of squares error. More intuitively, the least squares method attempts to find the line that minimizes the total squared distance between the line and all the data points [4 p. 5-12]. Linear models can also be utilized in classification, as was seen earlier with logistic regression.

While linear models are useful in very simple cases and they work as the basis of more complex models, such as support vector machines, they have shortcomings, which mean they are not practical for general purposes on their own. The curse of dimensionality causes the number of basis functions to grow rapidly when the number of input features grows, making linear models impractical for many real-life situations. While the effects of the curse of dimensionality can be alleviated to some extent, it is usually more practical to use more complex models, such as support vector machines or neural networks [4 p. 172-173]. Linear models also require the underlying linearity in the input terms to be present, which is not always the case [5 p. 43].

### 2.5.2. Nonlinear Models

Nonlinear models have the same goal in regression as linear models do: to model the predictive distribution of data so that it best represents the data and minimizes the loss function. The difference is that unlike linear models, nonlinear models do not meet the requirement of linearity in the parameters. In practice this means that the nonlinearity comes from the fact that the function cannot be expressed as a linear combination of the parameters. Also, while predictors in linear models only have a single parameter, predictors for nonlinear models can have multiple parameters affecting them. A simple example of a nonlinear model is the convex power function (7) where the predictor \( x \) has two parameters \( w_1 \) and \( w_2 \). Other examples of nonlinear functions are the Michaelis-Menten function for estimating enzyme velocity based on substrate concentration and many growth analysis functions, such as the Weibull growth functions [21, 22 p. 13-14, 47, 61, 23].

\[ y(x, w) = w_1x^{w_2} \quad (7) \]

While linear functions have one basic form, nonlinear equations have many possible forms. In fact, linear models are essentially a special variation of nonlinear
models. This flexibility makes nonlinear functions a more powerful tool for curve-fitting than linear models, which is why they are popular in biology and chemistry where most models are nonlinear in nature. On the other hand, the near infinite amount of possible functions also makes it harder to determine the optimal function for a specific case [22 p. 25, 47, 62-66, 23]. Nonlinear functions also generally lack the closed-form representation of optimal parameters, which means that fitting is usually done through successive approximations of parameters with numerical optimization algorithms [22 p. 91-96, 23]. Additionally, nonlinear function may have multiple local minima and the estimate for the global minimum might be biased [22 p. 37]. All this makes nonlinear models more difficult and demanding to work with, which is the tradeoff for their greater range of functions.

2.6. Decision Trees

Decision trees are non-parametric models that can be used for such predictive tasks as classification and regression. These structures employ a top-down approach, where binary splits are created starting from the root node until a stopping criterion is met. These splits are learned from the data and are based on the features and their values in the data. This is where the term tree comes from, as these models can be visualized as tree like structures. Decision trees are effectively a form of recursive partitioning, where a large space is divided into smaller regions until a complicated data space can be represented by simpler partitions. Each partition is essentially a smaller local model that is easier to handle than the whole data space [4 p. 663-66, 5 p. 305-312].

Decision trees are compelling for data science due to multiple reasons. They can handle both numerical and categorical data and multi-output problems. They are easy to interpret due to the decision rules being Boolean. The tree structures can also be visualized easily, as can be seen from Figure 1. This very useful for interpretation and visualization, which are very important, especially when people not familiar with machine learning are involved. Being non-parametric models, trees require little data preparation. As they essentially see ranks in features and partition rules are based on decision thresholds, standardization or scaling are unnecessary [4 p. 666, 5 p. 352].

The problem with singular decision trees is that they have caveats that make them less useful for machine learning. They tend to overfit and have high variance between training and test sets, which makes them perform poorly in general cases. In classification, bias can be introduced if some classes dominate the used training data. Trees also tend to be unstable, possibly leading to very different trees with small variations in training data. These problems can be reduced by combining multiple trees in ensemble models, which are called forests [4 p. 666, 5 p. 310-313, 352].

2.6.1. Classification and Regression Trees (CART)

A popular method for implementing decision trees is called CART, which comes from the term Classification and Regression Trees. CART is not the only method for tree-based regression or classification but it is probably the most common one and is
highlighted here. The general structure of a CART tree goes as follows. The root node is the first and the highest node, where the tree starts. It splits the tree into two regions based on the feature used at the root node, leading to two interior nodes. Interior nodes further split their region into two smaller regions with a binary decision based on their features. The splits can be thought of as questions, where each answer leads to a different new question. The trees end in leaf nodes, each of them representing a single partition of the larger data space. The leaf nodes contain a simple, local model that applies to that partition only, and the model depends from the nature of the tree. The resulting tree structure creates a Boolean decision rule set, which can be used to predict classes in classification or values in regression [4 p. 663-665, 5 p. 305-312].

Regression decision trees can be used to predict a continuous value, same as any other regression method. Constructing a regression tree is essentially a matter of recursively creating splits based on which feature can be split in a way that most reduces the sum of squares error of the tree. Another way to consider this is that the splits attempt to maximize the information gain at each node. The most impactful split with the most impactful feature creates the root node. Both resulting sub-regions are then split the same way. This continues until a stopping criterion is met or the tree is as large as it can be. The outermost nodes with no splits are the leaf or terminal nodes, that contain the local models of the partitions. For classic regression trees, the local model located at the leaf nodes is simply a constant estimate of the dependent variable. In practice this is the mean of the samples located at that partition of the data space that the leaf node represents [4 p. 663-665, 5 p. 305-308]. Another version of regression trees combines linear models and decision trees into linear model trees, where a leaf node contains a linear model instead of a constant value [24].

Because fully grown trees generally overfit to the data, while small trees might not necessarily model the data properly, tree size is an important tuning parameter for tree building algorithms. In practice optimal tree size depends on the data and should be learnt from it. A simple but naive approach is to have a threshold for the reduction of the sum of squares error that each split must meet and stop growing the tree when no new split meets this threshold anymore. The problem with this approach is that it is possible for a seemingly weak split to lead to a much stronger split after it. A more popular approach is to either grow the tree fully or grow a large tree, only stopping when a threshold of data points in leaf nodes is reached. This large tree is then pruned by moving upwards in the tree from the leaf nodes, collapsing leaf nodes based on their effect on the sum of squares error compared to the tree complexity. Cross-validation can also be used by using part of the data for training the tree and the other part to evaluate if collapsing some leaf nodes would improve the sum of squares error [4 p. 665-666, 5 p. 307-310].

Classification trees are very similar to regression trees, but they use a different measure, such as a misclassification error, for determining optimal splits in nodes. Their output is also a class instead of a continuous value. A simple binary classification tree with two features is shown in Figure 1, where c₁ and c₂ are classes and x₁ and x₂ are features. The blue nodes are binary decision splits based on values of the features. Node containing feature x₁ is the root node of the tree and node containing x₂ is an interior node containing a binary split based on the value of x₂. The green nodes are leaf nodes that decide the predicted class. A regression tree would look very similar,
except that the leaf nodes would either contain some continuous value or a linear model [4 p. 663-665, p. 308-310].

Figure 1. A simple binary classification tree.

2.6.2. Bagging and Boosting

Bagging or bootstrap aggregating refers to generating several new training sets from the original training set by sampling from the training set with replacement. Each new training set is used to create a model, and they are then combined by averaging their outputs for regression, or majority voting for classification. With trees this translates to creating multiple trees with different sets of training data and combining their results [5 p. 282-285]. These ensembles of trees are generally called forests.

Another way to combine multiple trees is through boosting. Boosting in general is a method of sequentially combining weak classifiers to produce a stronger classifier. The difference to bagging is that instead of creating independent models from different sets of training data, a weak classification algorithm is applied sequentially to repeatedly modified data for many iterations. Weights are applied and updated to observations at each iteration, and the final prediction is the weighted majority vote of each classifier. Boosting can also be used for regression. AdaBoost is a popular boosting method, which puts more weight on hard to classify observations, increasing their focus for the next iteration [5 p. 337-340]. For trees, gradient tree boosting is a popular method, where multiple trees are created sequentially, each tree trying to correct errors from the last one. This is essentially a gradient descent algorithm, and different variations of it can be derived using different loss functions. Popular loss functions include squared error, absolute error and Huber error, which is a compromise between the two [5 p. 358-361, 25].
2.6.3. Random Forests

Random forests are another way to deal with the weaknesses of singular decision trees. A random forest is a collection of tree structured classifiers or regressors, each of which are constructed from an independent random vector of samples and features based on the source data, but all of the trees maintain the same distribution. Random forest is essentially a form of bagging but using only a subsample of the feature space helps de-correlate the trees in the forest and improve performance by reducing overfitting. The output of a random forest model is the result of a vote between all trees. This can be the most popular class in classification, or the mean value in regression [5 p. 587-589, 26].

Random forests are a popular machine learning method due to their ability to measure the importance of a feature in data sets [5 p. 593-595, 26]. They also have good performance in both classification and regression, generally performing well even with little tuning. Random forests usually outperform normal bagging trees and are not far behind gradient boosting machines [5 p. 589-592, 26, 27]. However, being an ensemble of many trees, random forests are essentially a black box method and they lose the interpretability of decision trees. The decision logic used by a random forest model of any meaningful size is very hard to interpret. The way random forests measure the importance of a feature also introduces two kinds of bias, which can skew the results. This is discussed further in the evaluation and feature selection section.

Random forests have been used to predict performance in mobile cellular networks [28]. Random forests are also quite popular in the biomedical field, where they are used for classification and their ability for feature selection is used in the selection of genes, for example [29, 30].

2.6.4. Isolation Trees

Isolation trees are tree structures that can be used for anomaly detection. They operate on the assumptions that anomalous samples in data are few and they differ significantly from the normal samples, making them easy to isolate. They are based on randomly selecting features and then randomly selecting a split value between the minimum and maximum of that feature. This recursive partitioning can conveniently be expressed as a tree structure, where the number of splits that isolate a sample is comparable to the path length from the root node to the terminating node. This path length is the decision function for the isolation tree. Samples can be classified anomalous or normal based on this decision function, as random partitioning in theory creates shorter paths for anomalies. In other words, it takes less splits to identify a sample that differs from the norm compared to a sample that is within the norm. As is generally the case with trees, a single tree is not very useful, which is why a forest of many trees with different subsampling is generally built. For a forest, the average path length of all trees is used as the decision function [31, 32].

Isolation forests are computationally effective and have small memory requirements, as they have been found to quickly converge with small sub-sampling
sizes and do not require that many trees to be effective. They have also been shown to perform well compared to more traditional anomaly detection techniques [31, 32].

However, the assumption that normal samples have longer path lengths than anomalous samples is not universally true. This causes isolation forests to be blind to some anomalies. This can be remedied with hybrid implementation, where explicit knowledge of known normal and anomalous samples are incorporated into the model. This kind of hybrid method performs well in intrusion detection tasks [32].

2.7. Neural Networks

Neural networks are layered and connected network structures used for many machine learning applications. They are the main method for implementing deep learning and currently popular among the machine learning community. Neural networks are considered deep if they have more than one hidden layer, although the definition is not set in stone. Neural networks are based on the human nervous system, from where the name neural originates. They do not aim to completely model the brain, just to take inspiration from how it works [7 p. 164-167, 33]. Neural networks are not a new thing, but they were generally ignored for a while, because it was thought that they were impractical. Interest revived around 2006, when research showed that neural networks were not as unfeasible as thought [33].

2.7.1. Structure

Neural networks are composed of neurons that get their name and purpose from the neurons of a human brain. They are also called units. Like their biological counterparts, neurons in a network are connected to each other. Each connection has a strength, represented by its weight value. Each neuron also has a threshold value and an activation function. The activation function is a non-linear transformation that is applied between layers to the output of each neuron after they have calculated the weighted sum of their inputs. The most popular one is called rectified linear unit (ReLU). ReLU has become popular due its fast learning in networks with multiple layers and general applicability to most problems [7 p. 164-172, 33].

Neurons form layers. There are generally three kinds of layers. Input layer takes the input values and output layer gives the output of the whole network. The third kind of layer is called the hidden layer, where the learning takes place. They are called hidden as their input or outputs are not seen. Hidden layers can take different forms depending on their purpose in the network. A simple three layered neural network is seen in Figure 2. This network has three neurons in the input layer, four neurons in the only hidden layer and two output neurons. The network is fully connected, as each neuron is connected to all neurons in both preceding and following layers. It is also an example of a feedforward neural network, also called a multi-layered perceptron, which can be used to learn a function that can map a fixed size input to a fixed size output. Feedforward networks have no backpedaling, as the operations always go from input towards the output. These kinds of networks can be used for classification or regression [7 p. 164-177, 33].
Training of these networks can be done with simple stochastic gradient descent and backpropagation, where weights are adjusted after a few training iterations based on the outputs and errors. The errors are propagated backwards to previous layers, where it is used to help modify the weights. Weights are modified a little and a gradient is calculated with derivation for the new weights and each weight is adjusted towards the deepest gradient descent, i.e. where the output error decreases the most. This is iterated multiple times until improvement is no longer achieved or the improvement rate stagnates [7 p. 164-177, 33].

Key architectural questions for a deep neural network are its depth and the width of each layer. Depth refers to the number of layers that the networks has, and the width of a layer is the number of units in the layer. While more units and more layers can generally improve generalization, they tend to also be harder to optimize. Depth and layer width are tunable parameters that should be determined through experimentation [7 p. 193-197].

![Figure 2. A simple neural network with three layers.](image)

### 2.7.2. Deep Learning

Deep learning is a somewhat new methodology in machine learning that has gathered a lot of attention in recent years. It is generally understood as a solution that enables computers to learn and understand phenomena through concept hierarchy, where concepts are defined with their connections to other concepts. This hierarchy allows the computer to learn things from the bottom up, by forming higher concepts from simple ones. The name deep learning comes from the layered, deep graph that could be drawn to represent this hierarchy of concepts built by the computer [7 p. 1-2].
Because the feature layers are learned from the data, this way of learning has an edge over traditional machine learning methods, which tend to require hand-on feature engineering and domain knowledge [7 p. 2-5, 33]. This ability to learn and represent data with multiple levels of abstraction, combined with the vast amounts of data that has been gathering over the years, the advancements in computing enabling the use of more complex and larger deep learning networks and the development of better algorithms, have made deep learning the most promising topic in the machine learning and artificial intelligence fields lately. Deep learning solutions have already improved the state-of-art in many fields [33].

2.7.3. Applications

Neural networks and deep learning are suitable for tasks where an input vector can be mapped to an output vector, if there is sufficient labeled data and the models are large enough. Tasks that do not have this association between two vectors or would require a person to think and reflect before accomplishing it, are beyond deep learning [7 p. 163]. If these conditions are met, deep learning can provide a powerful tool, which is applicable for a variety of tasks. Many natural signals are compositional in nature, which means deep structures, such as deep neural networks, are excellent ways to represent them [33].

The field where deep learning probably sees the most use is computer vision and image processing for image classification and detection of objects in images. Convolutional neural networks are excellent for handling pixel images due to their ability to build representations in stages. They are a popular specialization of the deep feedforward networks that are designed to process inputs that come in multiple arrays, such as 2D pixel images [7 p. 326-339, 33, 34]. This is also the field that sparked the current deep learning boom in the industry in 2012. A convolutional neural network of eight hidden layers by Hinton et al won the ImageNet classification competition, improving the state of the art significantly and proving that deep neural networks had great potential for supervised learning tasks [33, 34].

Speech and language can also be modeled using deep structures. Deep neural networks work well together with the Hidden Markov Models (HMM) that are popular for modeling speech. Hybrid multilayer networks with both directed and undirected layers, called deep belief networks, have been used for speech recognition tasks to great success, outperforming the state of the art Gaussian mixture model counterpart [35]. Natural language processing has also benefited from deep learning. Recurrent neural networks (RNN) are another specialization of feedforward neural networks. They are meant for processing recurrent sequences like speech. What makes RNN special is that it processes each input one element at a time, while also maintaining information of the past elements [7 p. 367-388, 33]. Among other uses, RNN can be used for automatic language translation tasks. For example, an automatic English to French translation application was implemented with a neural network using LSTMs for both input and output vectors. The application performed well and was close to the state of the art [36]. Deep learning also sees use in the medical field and in chemical analysis, among other things [33].
2.8. Evaluation and Feature Selection

Regardless of what method or algorithm is used, its evaluation becomes relevant at some point. Generalization is a prediction capability of the method on test data that is not related to the data that was used for training. In practice this measure is important in assessing a model’s quality and choosing between different methods and models [5 p. 219]. How the performance of a particular method is measured in practice depends on the application and the model, and the choice is not always straightforward [7 p. 101-102].

2.8.1. Evaluation Methods and Measures

When a predictive model is created, its predictive power needs to be evaluated to see its usefulness. Typically, error rates between predicted and actual values are used for evaluation. These are usually called loss functions, and the aim is to minimize them for best performance. Absolute error and squared error with their variants, such as root mean squared error, are popular loss functions in regression, but not the only ones. They show how much the predictions differ from the actual values.

Let $n$ be the number of samples in some prediction task. $Y$ is a vector of the actual target values of the samples. $\hat{Y}$ is the vector containing the predicted values for the samples. Mean absolute error (8) is the average of the difference between the actual values and the predicted values. Mean squared error (9) is the average of the square of the difference between the actual and the predicted values. Using the square has the advantage of making gradient computing easier but it puts more emphasis on the larger errors than smaller errors, which can be undesirable [5 p. 219-222, 7 p. 106, 37].

\[
MAE = \frac{1}{n} \sum_{i=1}^{n} |Y_i - \hat{Y}_i| \tag{8}
\]

\[
MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2 \tag{9}
\]

For classification, accuracy is a popular performance measure that simply tells the proportion of correctly classified samples to all samples. While simple and intuitive, it can be misleading and not always the best measure. If the classes have uneven representation, a bad classifier could have high accuracy while always classifying samples as members of the more common class [7 p. 101-102, 37]. Other measures are needed to support it. A so-called confusion matrix provides a near complete picture of a model’s performance. An example of a confusion matrix for a two-class classification task with 100 samples belonging to classes $c_1$ and $c_2$ is presented in Table 1. The confusion matrix shows predicted classes in columns and actual classes in rows.
Table 1. Confusion matrix for a binary classification task

<table>
<thead>
<tr>
<th></th>
<th>Predicted $c_1$</th>
<th>Predicted $c_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual $c_1$</td>
<td>48</td>
<td>2</td>
</tr>
<tr>
<td>Actual $c_2$</td>
<td>3</td>
<td>47</td>
</tr>
</tbody>
</table>

Four terms are associated with confusion matrices: For class $c_1$, true positives refer to $c_1$ samples that were classified as $c_1$. True negatives refer to $c_2$ samples that were classified as $c_1$. False positives refer to $c_1$ samples that were classified as $c_2$ and false negatives refer to $c_2$ samples that have been classified as $c_2$. These terms can be used to calculate various performance features for a model. Precision can be thought of as the exactness of the classifier and is calculated by dividing true positives with the sum of true positives and false positives. Recall or sensitivity can be thought of as the completeness of the classifier, and equals true positives divided by the sum of true positives and false negatives. These two values can be used to calculate a F1-score that can be used to measure how precise and robust the classifier is. One other measure that can be calculated using the confusion matrix is specificity, which is false positives divided by the sum of false positives and true negatives. Using specificity and recall a receiver operating characteristic (ROC) curve can be plotted to see how the two interact at different classification thresholds. A ROC can be used to make decisions based on whether more false positives or more false negatives are preferred. The area under the curve (AUC) can also be used as a performance measure, with a higher AUC meaning more correct predictions for classes [5 p. 314-317, 37, 38, 39].

It should be noted that while confusion matrices can be done and are informative for visualising the amounts of correct and incorrect classifications for multi-class cases, the mentioned measures are for binary cases only. If these measures are required for a multi-class case, they have to be calculated on a per class basis. In practice each class is chosen once, and all the other classes are merged to one to form the two-class situation for the confusion matrix. These measures also have their own caveats, and care should be taken when using them to avoid false interpretations [39]. Some classifiers assign samples probabilities for belonging to each class instead of assigning them to one class. For these classifiers, logarithmic loss is a better performance measure, as it considers the uncertainty of the prediction. It calculates the mean log probability for each sample and each class based on whether the sample belongs to the class or not. The result is a value ranging between zero to infinity, with values closer to zero signalling better accuracy [4 p. 209, 37].

Generalisation error or test error is the desired value to estimate, as it is the prediction error of an independent test set. It can be hard to estimate test error from the training set and the expected test error is usually what methods estimate. Training error is the prediction error of the training set. It is a poor estimate for test error due to overfitting [5 p. 219-222]. Overfitting is the phenomenon, where the model fits the data used for training too well. This leads to great performance with the training data but can fail to perform in the general case. This is also called variance, which is sensitivity to small changes in the training set. Overfitting makes evaluation of the training set risky, which means that a separate set for evaluation is usually needed. [4 p. 32] Underfitting is the opposite of overfitting, where the learning algorithm fails to
learn the relevant relations between features and target outputs. This is also called bias. The bias-variance trade-off is a common dilemma in predictive modelling, where low bias leads to higher variance, and vice versa. The difficulty comes from trying to minimize both errors at the same time [4 p. 147-152, 5 p. 37-38].

The validation set is a portion of training data that is left aside from the training and then used for evaluation. This can also be used for choosing the best model when multiple models are created. With multiple iterations, even validation sets can lead to overfitting, so an independent testing set should be used for final evaluation of the model’s generalization [4 p. 32, 5 p. 219-223].

Because available data is in many real-life cases scarce and as much of it as possible should be used for training, big test and validation sets are not always possible. On the other hand, small evaluation sets can rarely evaluate the performance properly. One way to circumvent this is to use cross validation. It leaves part of the training data as validation and uses the rest for training. In k-fold cross validation, the training data is divided to k chunks. Each chunk in turn is used for validation, while the rest is used for training. In leave-one-out, each data sample is used for validation separately, making k the number of samples. This method has the advantage of using each sample both in training and validation, and each sample is used for validation exactly once. The downside of cross validation is that it can require many training runs and become computationally infeasible very fast, especially in situations with complex data sets or a large number of k. Also, when using subsets of features, properly applying cross validation is important to obtain correct results. [4 p. 33, 5 p. 241-247]

Error rates are not the only method used for evaluating models. Another way to measure the fit of the regression model is called the coefficient of determination, or R². It is the measure for the proportion of variance in the dependent variable that is predicted from the independent variables. It is calculated with the formula in (10), where $SS_{res}$ is the sum of the squared error and $SS_{tot}$ is the total sum of squares, which is the measure of variability in independent variable. When used properly it can range from zero to one, with the value of one signalling a perfect fit with the data and zero signalling that the independent variables do not explain the dependent variable at all [20 p. 31-32, 40 p. 97-100]. R² can have values outside this range, if a wrong model is used and the model fits the data worse than a horizontal hyperplane. The problem with R² in cases with more than one predictor variable is that there is an automatic increase in R² value when new explanatory variables are added to the model, regardless of the actual effect of the new variable on the fit of the model. This makes R² overly optimistic in multi-predictor cases. Adjusted R² helps deal with this. It adjusts R² with the number of samples n and the number of predictors k, as is shown in (11). It considers the degrees of freedom of the model. This means that if the effect of the added attribute on the fit of the model does not exceed the negative effect that adding new attributes has on the degrees of freedom, the value will decrease. If the attribute has a positive effect on the fit of the model that exceeds its negative effect on the degrees of freedom, the R² value will increase. Use of adjusted value of R² is recommended when there is more than one predictor variable [40 p. 102-103].
With machine learning, more data is generally better for training the model. It gives the method more to work with and allows it to better learn the patterns in the data. This does not apply to the number of features used for the training. While it might seem that adding more features would help better explain the patterns in the data, this is not always the case. Not all features influence the target values, and some can even affect the model’s performance negatively and introduce noise that makes it harder to model the relationship between relevant features and the independent variable. Many models, such as k-NN, also tend to suffer from higher dimensional data. In addition, using all the available features is not always feasible or even meaningful. Increasing the number of features increases the complexity of the model, which in turn generally leads to an increase in computational load [41].

The curse of dimensionality refers to the growing complexity of datasets when the amount of input variables increases. Higher dimensions become harder to illustrate and understand and make many methods unwieldy and impractical. This does not make higher dimensions unusable, as real data usually is confined to a region of the feature space, leading to smaller effective dimensionality. Real data also has smoothness properties which lead to small changes in target properties when small changes in input variables are observed, which can be exploited for analysis [4 p. 33-38].

Finding and choosing the optimal subset of features is called feature selection or feature engineering. It can be used to both find optimal feature sets with the best performance and to reduce dimensionality in the data set. Sometimes reducing dimensionality is desirable even if it decreases performance.

One common feature selection method is called sequential feature selection, which has two basic variants called Sequential Forward Selection (SFS) and Sequential Backward Selection (SBS). SFS goes through all the available features, starting from the most influential feature and always adding the next best feature to the set. SBS works backwards, starting with all features and removing the least significant feature with each step. A performance criterion, such as accuracy in classification, is used to evaluate and compare each feature set in terms of how it performs. Both approaches have a nesting problem: they cannot correct previous steps and remove or add already handled features. Sequential Floating Forward Selection (SFFS) and Sequential Floating Backwards Selection are attempts to correct this. They can reintroduce or remove already handled features if their inclusion or exclusion improves the performance. It should be noted that sequential feature selection methods are greedy algorithms and not exhaustive. They are not guaranteed to find the best feature set [41].
Another family of feature selection methods is called Genetic Algorithms (GA). GA methods take inspiration from natural evolution, and they perform randomised searches guided by a fitness measure. The result is a probabilistic algorithm which can be used for problems such as feature selection [41].

Another method for dimensionality reduction is principal component analysis (PCA). It is one of the most popular methods for dimensionality reduction and possibly also the oldest. The basic principle of PCA is to take the original observation data, extract the important information from it and project this new information into a new lower dimensional linear space and into new orthogonal variables called principal components. These new variables are obtained as linear combinations of the original ones. In addition to reducing dimensionality, this new expression of the data mostly removes the correlations between original features. This is useful as correlated parameters can affect prediction performance and affect feature importance estimations negatively. The downside of PCA is that it reduces the descriptiveness of the model, as the data of the original variables is lost [4 p. 561-583, 42].

In addition to reducing the dimensionality of the feature set, finding the underlying relationships between the input features and output targets is in many cases useful. Finding out which features influence the targets most can in itself be the aim of the analysis. Almost any real-life situation benefits from the understanding of underlying relationships between the features and between the targets and the features.

Tree structures allow for easy interpretation of the importance of a feature in data sets due to their nature. Both gradient boosting trees and random forests have a useful innate measure for calculating the importance of a feature within a data set. Called gini impurity, this measure is used to choose the local optimal condition based on which the decision to split the dataset in two in the nodes of the trees is made. In a regression case the impurity is essentially the variance or sum of squares error of the tree. This can be used to calculate the importance of a feature by computing how much each feature decreases the impurity in each tree and then using the average of the whole forest [5 p. 593-594, 43, 44].

While this ability is useful, it has some drawbacks. Two kinds of bias can emerge from this method. Firstly, when categorical variables are present, variables with more categories tend to be ranked higher than those with less categories [45]. This kind of bias can be corrected with heuristics [43]. Secondly, when correlated variables are present in the data, bias is introduced between them. When the dataset has two or more correlated features, it makes no difference for the model which of the correlated features is used as the predictor. For the impurity, this means that when one of them is used for the model, the importance of the others decreases significantly, as their effect on impurity has already mostly occurred with the first feature. This manifests as lowered reported importance [44, 45, 46]. The latter bias introduces a choice between the predictive and descriptive power of the model. The bias is fine when doing feature selection, as removing features that do not add much to the models’ performance is generally desirable, but it can lead to wrong conclusions when deciding which features are good predictors. If predictive power is valued, one option is to use PCA to project the features onto the principal components, to get rid of the correlation before training. If descriptiveness is desired, simply identifying highly correlated variables and removing duplicates is a simple method.
3. LONG TERM EVOLUTION AND THE TARGET ENVIRONMENT

The company that commissioned this work is a major international telecommunication and radiotechnology company that produces both telecommunications related hardware and software, such as radio base stations and their software. The company also develops other radiotechnology solutions and provides services and consultation related to telecommunications.

The unit that I was working in focused on the development and building of LTE standard compatible base stations and their software. More specifically, I worked with a team responsible for the LTE L2 capacity testing environment. The team creates and maintains test cases for testing and validating the performance of the products against the requirements set for them. Capacity testing is the main responsibility of this team. Although I was part of the team, this research was solely my responsibility.

Because specifics of the testing environment and the parameters used are not considered public knowledge, many aspects of the environment are either spoken of in very broad terms and using generic terms or excluded completely.

3.1. LTE

Long Term Evolution (LTE), also called the Evolved Universal Terrestrial Radio Access Network (E-UTRAN), refers to the advanced high-speed wireless communication standard of mobile devices such as smart phones. It was part of the 3rd Generation Partnership Project (3GPP) release 8 in 2008 and has since then become a stable for 3G and 4G networks [47 p. 11-13, 48]. In accordance to its name, LTE has been constantly evolving since its initial release, with 4G requirements fulfilling release 10 coming in 2011 to upgrade LTE to LTE Advanced [47 p. 11-13, 49]. Release 16 is currently under way and is supposed to fulfill requirements for an initial full 5G system [50].

The motivation for development of the 3GPP Universal Terrestrial Radio Access Network (UTRAN) LTE can be found in the evolution of the field. In addition to the obvious need for better performance, the evolution of the wireline domain required improvements on the wireless side to ensure similar standard of service. With the need to drive down the costs of data delivery and to keep up with the competition, E-UTRAN LTE was a necessary next step in development. Decreased latency, increased peak user throughput and more capacity with flexibility regarding new and old frequency bands were targets of LTE [47 p. 7-8, 51, 52 p. 4-5, 53].

Multiple new technologies were adapted to LTE to achieve the new target data rates. LTE uses Orthogonal Frequency Division Multiplexing Access (OFDMA) for downlink traffic and Single Carrier – Frequency Division Multiple Access (SC-FDMA) for the uplink traffic [51, 52 p. 5, 53, 54]. OFDMA was chosen because it can achieve the desired high data rates with low cost, power-efficient hardware and relatively simple solutions. OFDMA technology also enables higher network bandwidths of up to twenty MHz by splitting the channel into multiple narrower sub-channels. Older Wideband Code Division Multiple Access (WCDMA) networks were
limited to five MHz channels, due to wider channels causing delay problems. SC-FDMA resembles OFDMA technically but is less demanding for battery power in mobile devices, which is important for consumer satisfaction. Other new technologies in LTE are multiple input / multiple output (MIMO) and high rate modulation schemes. MIMO is used to lessen the effects of noise in a network by sending data in multiple sub-streams to increase possible data rates and throughput. Modulation refers to how the data and transport wave are combined, and LTE has multiple techniques for modulation for both uplink and downlink to achieve the high data rates [53]. In addition, LTE also changed to a fully packet switched network in both data traffic and voice, compared to the circuit switched voice handling of previous networks. Voice calls in LTE use Voice over Internet Protocol (VoIP) with packet-based sessions [48 p. 7-8, 51, 52 p. 23-24, 54].

### 3.1.1. LTE Architecture

Simplified architecture and removal of unnecessary options were set as goals for LTE. New architecture was streamlined by reducing the amount of network elements and separating the user plane and control plane. This resulted in a two-part architecture with the core network being called the Evolved Packet Core (EPC) and the Radio Access Network being called E-UTRAN. Together with User Equipment (UE) they form the Evolved Packet System (EPS), which connects the UE to external networks and services such as the Internet. One example of a typical LTE high-level architecture with its various components and connections is shown in Figure 3 [47 p. 109-110, 51, 52 p. 23-26, 53, 54].

![Figure 3. High level architecture of a typical LTE network.](image)

The UE refers to devices that the end users use to communicate with the network. These devices can be any device capable of connecting to the network, such as smart phones or data cards embedded in devices like laptops. They offer the end users both a platform to communicate with the network and a user interface for the user to make
use of the various applications such as voice calls. The users are connected to the network through E-UTRAN [51, 52 p. 26-27, 54].

The E-UTRAN is only comprised of Evolved Node B radio base stations (ENodeB). With no central control node, the architecture of E-UTRAN is flat. E-UTRAN handles all radio related functions, including radio protocols, data delivery, security, mobility management, retransmission and header compression. The ENodeBs are connected to each other through a X2 interface. X2 enables handovers between nodes to support mobility. ENodeBs are connected to the EPC through the S1 interface. The connection is split between the user plane and the control plane. S1 user-plane part S1-u is connected to the Serving Gateway (S-GW) and S1 control plane part S1-c or S1-MME is connected to the Mobility Management Entity (MME). An ENodeB can be connected to multiple MME and S-GW at the same time, but each UE can only be connected to a single MME and S-GW at a time, which the ENodeB has to keep track of [47 p. 110-111, 51, 52 p. 27-28, 53, 54].

The MME is the main control component of the EPC and is not involved with the user plane at all. It is responsible for authentication and security of the UEs, tracking UE locations for mobility management and handling subscription profiles and service connectivity. MME also handles bearer management. Another component called the Home Subscriber Server (HSS), helps the MME perform its responsibilities. The HSS is a database server that stores the permanent subscription and location data of the registered users [47 p. 110, 51, 52 p 29-34, 54].

The System Architecture Evolution Gateway (SAE-GW) serves as the user plane connection to the external networks. The SAE-GW is split into two parts. The S-GW serves as the user plane connector between the ENodeBs and the EPC. It relays the data between the ENodeB and the Packet Data Network Gateway (P-GW) and acts as a mobility anchor when the UE switches ENodeBs. Additionally, it collects data and statistics for charging. The P-GW acts as the connector between the EPS and the external networks. The P-GW is also a mobility anchor that acts as the Internet Protocol (IP) point to the external networks for an UE. The P-GW generally assigns the IP that the UE uses to communicate with the external networks. It also performs traffic gating, information gathering for charging and quality of service enforcement. The Policy and Charging Function (PCRF) is responsible for Quality-of-Service (QoS) and charging. It is connected to both the P-GW and the S-GW and informs them of policies and QoS requirements for services [47 p. 110, 51, 52 p. 30-33, 54].

### 3.1.2. LTE Radio Protocols

The LTE radio access protocols are handled by L1 and L2 for both the user plane and the control plane. Control plane radio protocols also extend to L3 with Radio Resource Control (RRC). The three Ls refer to the first three layers of the Open Systems Interconnection Reference Model (OSI-Model): the Physical layer, the Data Link layer and the Network layer. These protocols handle the signalling for both the user plane and the control plane and enable the connection between the UE and the EPC. Signalling for both planes is essentially packet data, with the user plane handling the end user IP data traffic while the control plane handles mobility and other signalling
related to creating and maintaining connections. The user plane protocol stack is shown in Figure 4 and the control plane protocol stack in Figure 5. Only the protocols from the UE up to the MME for c-plane and up to the S-GW for u-plane are shown. The protocols relevant for this work are highlighted with green colour. The c-plane also has a direct signalling transport between the UE and the MME called Non-Access Stratum (NAS), which is not visible to the ENodeB. The NAS and other protocols not highlighted are not relevant to this work [47 p. 111-113, 51, 52 p. 35-39].

Figure 4. LTE user plane protocol stack.

Figure 5. LTE control plane protocol stack.

An example flow of data in an ENodeB in a downlink case with three IP packets for the user plane is shown in Figure 6. The flow and relevant protocols are also explained in the following paragraphs. The data flow is similar for the uplink but into the opposite direction. It is also similar for the control plane but includes the RRC in L3, which is included in the figure for reference. The RRC is located in an ENodeB and controls the connection states of the UE in E-UTRAN. It is a major control component in E-UTRAN. The RRC also handles handovers, Radio Bearer (RB)
management and many other control functions and procedures [47 p. 111-125, 51, 52 p. 137-148].

Figure 6. LTE downlink data flow through radio protocols in an ENodeB.

Data is received in lower layers from higher layers as Service Data Units (SDU). After receiving an SDU, a layer performs its services on it, building a Protocol Data Unit (PDU) that it then sends to the next layer below it. In LTE, one or more RBs are provided for UEs to work as tunnels through which data traffic goes and to which IP packets are mapped based on their QoS requirements. Data Radio Bearers (DRB) carry user plane data, while Signaling Radio Bearers (SRB) handle control messages such as RRC messages [47 p. 111-114, 52 p. 137-138].

The Packet Data Convergence Protocol (PDCP) layer receives IP packets from the upper layers. The PDCP is responsible for header compression/decompression of IP packets. Robust Header Compression (ROHC), a standardized header compression algorithm used in many telecommunications applications, is used. Compression is
important to reduce overhead to allow for better data rates, especially with smaller IP packets. The PDCP also handles security with ciphering/deciphering. Integrity protection and verification are also handled in the PDCP to ensure the correct source. In the example data flow the PDCP performs header compression and ciphering on the received SDUs and adds a PDCP header with information for deciphering before forwarding the resulting PDU to Radio Link Control (RLC) [47 p. 111-114, 51, 52 p. 145-147].

The RLC layer receives the SDUs from the PDCP. The RLC performs various operations on the SDU depending on its mode of operation. In transparent mode the RLC simply forwards SDUs between other layers without adding any headers. In unacknowledged mode the RLC performs most of its functions, including in-sequence delivery of data that might have been received out-of-order, segmentation and concatenation of SDUs into RLC PDUs and general protocol error correction. Concatenation and segmentation allow for dynamic sized PDUs and are shown in the example figure. In-sequence delivery is achieved by adding the sequence number to the headers of each PDU. In acknowledged mode the RLC performs the same actions as in unacknowledged mode, but additionally retransmits lost PDUs if the Hybrid Automatic Repeat Request (HARQ) is in use and detects duplicated parts. In the example data flow, the RLC performs both concatenation and segmentation, adds headers with sequence numbers for in-sequence delivery and retransmission and forwards the resulting RLC PDU to the Medium Access Control (MAC) through logical channels [47 p. 113-115, 51, 52 p. 143-145].

Logical channels allow the MAC layer to offer services to the higher layers. Logical channels are divided into traffic channels that carry user data in the user plane and control plane channels that carry control and configuration information. There are many further types of logical channels defined by the type of data they carry, but all variations are not listed here. The MAC itself handles multiplexing/demultiplexing and the padding of received SDUs into MAC PDUs, also called Transport Blocks (TB). The MAC is also responsible for scheduling in the ENodeB to prioritize traffic between the UEs and the logical channels. The MAC also handles error correction through retransmissions with the HARQ and is responsible for traffic volume measurement and transport format selection. Additionally, the MAC handles multiplexing of logical channels and mapping logical channels to appropriate transport channels and is responsible for handling multiple component carriers in carrier aggregation. In the example data flow, the MAC takes the SDUs sent by the RLC, multiplexes them, adds a MAC header containing the identities of the logical channels where the RLC PDUs originated from alongside other control information and forwards the resulting TB to the Physical Layer (PHY) through the transport channels [47 p. 115-118, 51, 52 p. 139-143].

Transport channels allow the PHY to offer its services to the upper layers. There are multiple different transport channels, which are defined by how the information is transmitted. Data in transport channels travels in TBs, with a Transmission Time Interval (TTI) having at most one TB being transmitted over the radio interface per UE. MIMO can increase the amount to two per TTI. TBs have a transport format specified by the MAC, which defines how the TB is transmitted over the radio
interface. As the transport format affects the data rate, the MAC can control data rates with it [47 p. 116-118].

The PHY maps the transport channels to physical channels. A physical channel corresponds to a set of time-frequency resources used for transmission of a specific transport channel. The PHY heavily influences the resulting capacity and resources of the radio access system. It is responsible for coding/decoding, HARQ processing, modulation and mapping of the signal to the necessary time frequency resources. In the example, the PHY receives the TBs from the MAC, adds a Cyclic Redundancy Check (CRC) to them, performs modulation and coding to them and transmits them to the physical channels. Multiple antennas might be used for transmission if MIMO is in use, in which case the PHY handles the mapping procedures [47. p. 113, 123-123, 52 p. 83-93 141-142]. The CRC acts as an error detection part in the receiving side to decide whether the decoded bits are correct [47 p. 113, 52 p. 353].

### 3.2. Testing Environment

The software development at the company follows Agile methodologies. These methodologies are generally known for dividing the development process into small, short iterations, with focus on working code over documentation. Person-to-person communication is also highly valued in Agile, and regular in-team meetings are encouraged, where each team member tells others what they have done since the last meeting and what they plan to do next [55].

The open-source automation server Jenkins is used in the company to maintain and automate the testing environment, and a version control system is used to maintain the code base. Both Git and Subversion are used for version control. Jenkins houses and controls the tests that are run to evaluate and maintain the code quality and integrity. Tests also evaluate the various performance measures set for the system. Tests are run as jobs, which can contain one or more test cases that are run sequentially.

Continuous integration is practised in the company by having developers submit their code commits, called revisions, into their own branches in the version control system, which are then merged into the main code line when deemed suitable. Commits generally cause jobs in Jenkins to build and run on the new code, to ensure code integrity and to evaluate the effect of the commits on performance. Keeping it green is the responsibility of all developers, which means that all commits should build correctly, and all tests should pass. Not all jobs are run with every commit, as some choice jobs are instead run periodically on the latest successful revision. If the commit causes problems, the version control system allows for easy rollback to a previous, working revision. The version control system also keeps a history of commits, allowing for later review if problems manifest with delay [56].

On top of maintaining code integrity, the main purpose of the testing system is to test that the software and hardware are suitable for production and release. Many different types of testing are done in the testing environment, but this work is focused on the capacity testing portion of the system.

Capacity testing aims to evaluate whether the artefact under testing can handle the traffic that it is designed to handle [57]. Capacity testing is also a part of System
Component Testing (SCT), as the tested artefacts in the company consist of multiple components and functions. SCT tests components separately to verify their functionality. Some functionality of the component might require interaction with other components, in which case these components have to be included in the testing. If the components are not available or even developed yet, stubs and drivers can be used to simulate their behaviour to a degree that allows testing of the necessary functionality [58].

The capacity tests in the company are meant to test whether the products meet the requirements set out for them, and to help in finding possible bottlenecks if requirements are currently not met. Testing is performed on real hardware. The system that is being tested is called the System Under Testing (SUT) and is the target of interest and evaluation in testing.
4. DATA MINING AND PARAMETER DEFINITION

The eventual goal of this work was to study estimating SUT CPU loads from L2 test parameters and to find parameters that influence it the most using random forest regression. This was not the original goal, as the effort started with a more open-ended objective. The goal materialized through exploration of the target environment and available data.

The company originally wished to research the possibilities of applying machine learning algorithms into their testing system and automating performance monitoring. Specifically, this work was aimed at monitoring the capacity test cases. As the number of test cases grows and more and more data is produced, handling the results becomes harder and takes more time. An automated tool that would evaluate the results and point to likely problematic jobs would allow the developers to focus on fixing the problems instead of searching for them, making work more efficient.

An additional aim was to explore machine learning in general and determine how well the target environment supported data science. As there are many kinds of data related to the results of tests and many kinds of machine learning algorithms, part of this work was to determine what data to use and which machine learning algorithm would be applicable.

A big and important part of any data science project is defining the data. Data is rarely readily available in a neatly organized and immediately usable format, especially in real life situations. This was the case for this work, and a major portion of the work went into defining available data, deciding which data was useful, extracting the data and analyzing the viability of the data and which methods to use. This effort is highlighted in this and the following section, explaining the process of examining the available data, analyzing it with different methods and finally coming to a decision of what to ultimately use in the work. The process flow of this work is also visualized in Appendix 1.

4.1. Defining Available Data

The first step of this work was finding and defining the available data. Being a well-established telecommunications company with years of history, the amount and variety of raw data was massive. The testing environment present at the company provides a multitude of tools that visualize and present the various data available. These tools are used by the testers to track performance. Part of the reason why there is a drive to apply machine learning and automation into the process is to automate the analysis of the results and free up resources for other tasks. While these tools in themselves do not have the data available, they give an idea of what kind of data is available by showcasing it.

In this work, only numerical data was of interest, so any text data was not considered. Specifically, data related to test performance, test parameters or test results was sought after. Browsing the various tools and views of the testing environment showed that there is plenty of different statistics stored in various formats within the
company’s systems. Looking into where the data was stored revealed databases that were a good start for data exploration.

4.1.1. Databases

The first obvious source of data was the various Structured Query Language (SQL) databases that the company was maintaining. Databases are excellent for data science projects, as they store data in an organized format and SQL databases have well established ways to extract data from them. These databases mostly held various results for the test cases, including the CPU loads for various jobs run in Jenkins. The data from databases was simple to get as JavaScript Object Notation (JSON) files. The JSON-files were then easily parsed into .csv files, as JSON is a common structured file type with plenty of resources for handling.

Unfortunately, the problems with the data in databases quickly became apparent. The data was from multiple sources, which made it inconsistent in parts. Different data had also been gathered for different amounts of time, making comparison harder. These factors meant that after data engineering the various data into a comparable and usable form, there was not that much data left. This was problematic, as most machine learning solutions rely on a good and sufficiently sized data set to be taught on. An additional problem was that the databases only stored test results, making analysis of what caused the said results harder.

Still, some jobs had a long history of data stored in the databases and it was considered interesting to see whether analysing long job trends would provide useful insights. Additionally, Python was the choice of development language for this work. This meant that there was easy access to ready to use third party implementations of many popular machine learning algorithms. This enabled quick and easy analysis that allowed exploring different options without investing much time and effort.

4.1.2. Job Artefacts

Each job produces a wide variety of different artefacts as results when they are run. These artefacts contain relevant information about the job, the test run and the results of the run. The previously explored databases for example get their data almost exclusively from these artefacts.

Being the source of most of the data in the system, these artefacts became another focus for data exploration. While a lot of the artefacts were not useful for this work, there were text logs that outlined the test runs from start to finish. They contained all the results that the tests produced and had the various parameters and test setups that defined each test. It was eventually decided that these logs would be used as the source of data for this work, even though recovering the data and modifying it into a usable format would require extra work. These artefacts were obtainable using common Universal Resource Identifier (URL) queries, which was easy to script and automatize.

The problem with using the job artefacts as a data source was that they required active data gathering and handling. While the data in databases was easily obtainable and persisting, the job artefacts had to be manually stored and the data from them
parsed. Another problem was that fetching older logs was not feasible, which meant that available data effectively started from near zero and data gathering had to be begun. The advantage of job artefacts as a source of data was that they contained both test results and the test parameters, allowing for more possibilities with the analysis and more flexibility in choosing what data to use. Using them also allowed for a more complete scope of data, as all jobs produce these artefacts. This was important, as all necessary jobs could now be included in the analysis.

4.2. Test Case Parameter Definition

SUT CPU loads are one of the performance metrics tracked in the testing environment. When it was decided that job artefacts were to be used as a data source, focus started to shift towards analyzing CPU loads. As the job artefacts had both the various test parameters and the CPU loads in them, their relationship was a logical target of research. There is also some previous research into their relation suggesting that there is a meaningful avenue of research available. As there are many different applicable parameters that could be considered for analysis, determining the most universal ones that are common for all test cases and are available from the test logs was important. The parameters are related to the PDCP and RLC layers discussed in the LTE radio protocols section of this thesis.

4.2.1. Choosing Parameters

For the data in databases, the choice of parameters to use for analysis was straightforward. Job run duration, CPU loads and the success rate were chosen as those were the ones that apply to all tests and are relevant in evaluating the quality of a test run. Success rate refers to the amount of cases that succeeded in the run compared to the total number of cases in the job.

As for the test parameters from job artefacts, the initial parameters to be used in the analysis were chosen through expert knowledge and previous research. Maintaining CPU loads at an acceptable level is an important part of the development of the target system, and those working on it have developed a general idea of what affects it. Their input was used as a basis for the choice of parameters.

Additionally, there have been previous efforts to estimate performance and what affects it in this system. Kananen studied the effects of RLC and PDCP parameters on the CPU loads in the C-RAN (Cloud Radio Access Network) environment, and created a tool based on a constructed regression model to estimate loads and the needed number of CPU cores with different values of the parameters. He also identified the importance of different parameters. Out of the eight parameters that he originally used he was able to pinpoint the five most important ones [59]. His findings were used to support the expert knowledge when choosing the parameters. Additionally, his findings that the relationship between the parameters and the CPU loads might not be linear in nature was used to rule out linear methods from the candidate methods to be used in this work. Another effort for parameter and performance estimation was done by Holappa, who used statistical analysis to determine L2 processing latencies in an
LTE eNodeB [60]. While the analysis is not directly comparable to this work and is somewhat dated now, it provided some insight into the system and provided help on how to approach work in this environment.

In total eleven parameters were initially chosen to be used in the analysis. For a sample size of n, the parameters form an input vector $X$ of size $(n \times k)$, where $k$ is the number of features. Similarly, for a sample size of n the observations form an output vector $Y$ of size $(n \times 1)$. The vectors are presented in (12). Due to confidentiality, the parameters are not named. This does not matter, as the names are not relevant for the analysis. Dummy names ranging from $x_1$ to $x_{11}$ are used when they are mentioned.

$$
\begin{align*}
X &= \begin{bmatrix}
    x_{11} & x_{12} & \ldots & x_{1k} \\
    x_{21} & x_{22} & \ldots & x_{2k} \\
    \vdots & \vdots & \ddots & \vdots \\
    x_{n1} & x_{n2} & \ldots & x_{nk}
\end{bmatrix},
\quad Y = \begin{bmatrix}
    y_1 \\
    y_2 \\
    \vdots \\
    y_n
\end{bmatrix}
\end{align*}
$$

4.2.2. Log Parsing

The L2 test parameters for test cases were not readily available in a usable format. While the specifications and definitions of test cases were present in multiple places and in multiple formats, most of them did not support data extraction and for that reason were not useful for this work. This is understandable, as the system was not designed with machine learning or data science in mind. Machine learning research and the need for storing and extraction of data in an easily usable format are recent developments in the company. The best source for the parameters were the test case result logs. They had all the test parameters in somewhat a consistent form and were data mining friendly. Similarly, they had the CPU loads and the necessary test case metadata.

Scripting was still required to transform the data into a usable format. A script was developed using Python that parsed the parameters from the logs.Parsed parameters were stored in .csv files, which are commonly used as the data format in data science and machine learning projects. Most Python data science libraries are compatible with .csv files as is. In addition to the eleven parameters and the CPU loads, some additional metadata was parsed for manual analysis and comparison. This included the name of the test case, the name of the job it belonged to, the code revision and the timestamp of the run. The parser outputs .csv files on a per job basis, with each .csv file containing data of one job. The parser either creates a new .csv file if one does not already exist for the job or otherwise appends to an existing file.
5. EXPLORATIVE ANALYSIS

To determine which data and which methods were useful, an explorative analysis was conducted. Before the analysis, data that was not obtainable from the database was gathered from the test result logs for a week. Jupyter notebooks were used as the development and analysis platform with various open source Python libraries providing the necessary machine learning tools. These notebooks are part of project Jupyter\(^2\), an open-source effort to support interactive scientific computing and data science across programming languages. The notebooks are essentially documents that can contain live code alongside visualizations and explanatory text. These notebooks can be shared like any other document through email or file sharing sites to distribute code and results. The notebooks are accessed with a browser, which also enables them to act as remote development platforms.

The data was analyzed with several machine learning and data mining methods in order to determine what methods could provide meaningful results. The analysis was explorative in nature, meaning that the methods were unoptimized and the aim was to see if any method could produce useful results on the data at hand. The exploration was divided into two distinct parts based on the data used. First the data from databases was analyzed, and then analysis was done on the data extracted from test result logs. The test setups, results and decisions made during the exploration are presented below.

5.1. Python libraries

One wish of the company was that the implementation would be easily portable and easily run on most machines. They also suggested Python as the preferred coding language of choice, as it was already in use at the company and installed in most machines. Python is also easily installed on most operating systems. Additionally, Python’s extensive third-party libraries and existing machine learning implementations made it perfect for this work. There has been an extensive push to make Python a viable alternative in data science for more traditional tools like R and MATLAB. Part of the explorative analysis was to test the viability of various libraries for this work.

Most machine learning implementations in this work are from the scikit-learn Python library. It is a machine learning resource created in collaboration by many different international volunteers. It makes tools for data analysis and data mining easily available for layman through its Application Programming Interface (API) and can be used for data science and machine learning tasks [61]. The version used in this work is 0.19.1, the most recent at the time of starting this project. The library received

\(^2\) http://jupyter.org/ is the homesite of Project Jupyter.
updates during the making of this work, but those were not applied to the working environment to ensure that no conflicts arose.

Another machine learning library used in this work is MLxtend. It implements select core algorithms for machine learning and data mining for use in day-to-day data science tasks [62].

Data structures in this work are mostly handled with Pandas, which is an open source Python library providing data structures and data analysis tools for use in data science. Pandas’ biggest contribution is its data structures Series and Data Frame, which allow versatile storage and transformations of labeled data. Pandas data structures are optimized for data analysis and modeling, and they are supported by both scikit-learn and MLxtend [63]. The Pandas version used in this work was 0.23.

As Python is somewhat lacking in its ability to do scientific computations and some mathematical operations, NumPy was used to provide necessary functions. Like Pandas, it provides more flexible data structures and data structure manipulations more fitting for scientific calculations and basic scientific operations useful for data analysis [64]. It is compatible with the previous three libraries and is recommended to be used in conjunction with them. The version used in this work is 1.14.3.

Finally, data visualization is done with Matplotlib, a free Python library that provides a wide range of tools for visualizing data [65]. It works with the previous libraries, and for Pandas and scikit-learn it is the recommended way of visualizing data, as those libraries have no in-built visualization mechanics. The version used in this work is 2.2.2.

5.2. Database Data

As one of the goals of this work was to create a tool to track problems in the testing environment based on CPU loads, anomaly detection and clustering methods were explored. Three algorithms were applied to the historical data of a single job: isolation forest, one class SVM and $k$-means clustering. All of these can be used for unsupervised machine learning to detect patterns from the data. All algorithms were applied through implementations from the scikit-learn Python library. Default hyperparameters of the implementations are used unless otherwise stated.

5.2.1. Test Setup

A single job with a long history of data was chosen as the target of analysis. Parameters used in each sample were the CPU load, job run duration and success rate. Revision of each sample was used for visualizing the development of parameters over time.

Isolation forest was run with 1000 trees and 256 samples per tree, as was found optimal in the original Isolation Trees paper [31]. Contamination, i.e. the excepted proportion of anomalous samples was set to 0.1. One class SVM was run with the radial basis function kernel, and nu and gamma values were set to 0.26 and 0.5, respectively. Nu refers to the upper bound of the fraction of training errors and the lower bound of the fraction of support vectors. Gamma is a kernel coefficient. $k$-means
clustering was run with values of k ranging from two to seven, and the optimal value was chosen based on inertias.

An interesting aspect of the data was that it featured concept drift. Concept drift refers to a change of trends in input features over time. It causes changes in the probabilistic distributions of features and usually degrades performance of models. It can be alleviated by retraining models and forgetting older information after it is no longer relevant to the current trend. Also adding bigger weights to newer samples and decreasing weights on older samples can help [66]. This shift in result trends was caused by changes in the case parameters as testing requirements changed. Concept drift can negatively affect machine learning and had to be taken into consideration while analyzing results. In practice the timeframe of samples was constrained to one consistent trend if concept drift seemed to affect the outcome.

5.2.2. Results and Discussion

Of the three methods, isolation forest showed the most promise. It was seemingly able to isolate anomalous runs from the data. Interestingly, isolation forest was seemingly able to deal with the concept drift, generally marking samples that differed from the current result trend as anomalous. Figure 7 shows the results of anomaly detection with isolation forest, with red dots representing anomalies and green normal samples. It also shows the concept drift over time in the data set. Isolation forest clearly marks most loads differing from the current prevalent trend as anomalous. The red dots seen within normal loads mostly represent samples that had differing job durations or success rates from the norm.

Figure 7. Results of anomaly detection with isolation forest on historical data of one job.
One class SVM however fell victim to the concept drift, and instead of pointing out anomalies in the overall data, it took the middle ground of all test trends as the norm and marked everything else as an anomaly. This is shown in Figure 8, where the results with all data is shown. Limiting the data to a timeframe with a constant test trend showed somewhat better results, with one class SVM showing separation between normal and anomalous loads, as is seen in Figure 9. The results were still inferior to isolation forest, and the need to exclude a large part of the data was a major negative.

Figure 8. Results of anomaly detection with one class SVM on full data set. Effects of concept drift can be seen.

Figure 9. Results of anomaly detection with one class SVM on limited data set.
Using the visualized inertia values four was decided to be the optimal cluster amount. It was also a logical division considering the amount of input variables. Concept drift also affected $k$-means, with the algorithm clustering samples based on load trends in the whole data set. This can be seen from Figure 10, where one trend was selected as the blue cluster, and the rest either as green or purple. Selecting a timeframe with consistent result trends similar to the one class SVM produced better clusters based on different loads and durations. In Figure 11 one can see better clustering. The green, normal samples make the largest cluster, while samples marked red have higher than usual durations, and the purple and blue samples have worse than usual success rates and purple samples also have lower CPU loads.

Figure 10. Results of $k$-means clustering with four clusters showcasing effects of concept drift.

Figure 11. Results of $k$-means clustering with four clusters and limited timeframe.
While the analysis showed some promising results, it was not enough to warrant further analysis in the scope of this thesis. While there were several jobs that had enough data to do successful anomaly detection of test results, and both isolation forest and k-means showed great promise in being able to detect anomalous runs, it was not enough for the tool that was planned. As the aim of the tool was to track performance of all of the tests measuring capacity, only relying on the data in the databases would not be enough. The amount of data was unlikely to be enough for reliable results, and the developed tool required more data. This meant that the data from the databases was not used further in this work.

While the data in the databases was not enough for this work, it should be highlighted that the results for the one job that was analysed were very promising. Isolation forest performed very well at marking samples with anomalous values in one or more of the input features. Similarly, $k$-means was able to cluster samples based on whether they were normal or had anomalous values in one or more of the input features. This kind of analysis should be considered if the databases are ever expanded to store results of more jobs. Even with the jobs already in the database it might be useful to track their performance with isolation forest or $k$-means, even if the scale is not enough for this work. Further development and expansion of the databases is however needed for them to be useful in more extensive performance tracking using machine learning.

5.3. Job Artefact Data

Three different configurations for the test parameters and loads were used in the analysis. The reason for this was the ambiguity of how the features should be represented. The L2 architecture and LTE radio network structure makes representing the feature space non-trivial for more complex test cases. Additionally, some test cases test multiple SUTs at once, further complicating matters. Using multiple variations of the features was an attempt to see if any provided clearly better results than others. These three variations of how the data was represented were all evaluated separately but with the same process to determine which one obtained the best results.

5.3.1. Test Setup

All algorithms mentioned here were applied through implementations from the scikit-learn Python library, unless otherwise stated. Default hyperparameter values for the implementations are used unless otherwise stated. All algorithms for every configuration were run with the same seed number (50) for the random number generator to keep results consistent and comparable.

To gather a training data set for analysis, logs of the latest successful runs were collected for five days and parsed for the data. Additionally, one more batch was collected on the sixth day to be used as a test set. 178 jobs provided usable data, with 546 cases in total being represented in the data. It should be noted that not all cases provided usable data every day, as failing cases rarely provide the data needed for this
analysis. Results that were not useful for the analysis were filtered out from the final data sets. Due to a human error, the training sets for the three models were uneven. The training sets were sized 3034 samples with 107 failed cases for configuration 1, and the two other configurations had 2613 samples with 92 failed cases. Essentially this difference amounts to one days’ worth of samples. While this discrepancy certainly affects the comparability of the models negatively, it was deemed acceptable due to the explorative nature of the analysis.

Isolation forest and one class SVM algorithms were applied to the training sets. The goal was to see if anomaly detection could be used to detect problematic cases from the training set. Isolation forest was run with 10,000 trees. 256 samples per tree were used, which is generally considered the optimal number of samples for isolation forests. Bootstrapping was used to train each estimator by random sampling with replacement from the training set and all features were used for each estimator. For SVM, the parameters were mostly left to their default values for the implementation. Linear kernel was used, and the nu value was set to 0.1. For these two algorithms the CPU loads were included in the input vector.

Clustering was also applied to the training set for each model. K-means clustering was used to see if meaningful clusters could be identified from the data. The data set has two categorical features, which standard k-means is not usually suitable for handling, but clustering was still considered worth looking into and k-means is the first go-to method for clustering. The training set was standardized by centralizing and scaling to unit variance by dividing features by their standard deviation for each feature independently. This was done as k-means suffers when features are uneven and of different units, which is the case in this situation. Cluster amounts from one to fifteen were tried, and the inertia of each cluster amount was visualized. While inertia has its flaws as an indicator of the right number of clusters, it can still be used as a guideline of which cluster amount provides the best cluster split. Based on the inertias, the number of clusters were chosen and the results were visualized and analysed. The goal was to see if clustering could provide interesting insight into the feature space, and to see if clustering could be used to identify anomalous runs.

To explore the dependencies between features and to create a model that modelled the relationship between the L2 test parameters and the CPU loads, regression algorithms were applied to the training set and then evaluated with the test set. Failed cases were filtered out from the training and test sets to make the data clean without noise. The training set for configuration 1 was 2912 samples after removing duplicates and failed cases. For configurations 2 and 3 the equivalent set was 2506 samples. The test sets used to evaluate the models were sized at 530 samples for all configurations.

Two different regressors were used on the training set. Random forests are a non-parametric ensemble method for both classification and regression. They can also be used to estimate the importance of features, which was also done in the analysis. 1000 trees were used, as more trees did not improve the results in pre-testing. For comparison, regression with neural networks was also chosen to be tested on the same training and test sets. This was done to see if neural networks were a viable option for automatic performance monitoring in the system. Multi-layered perceptron (MLP) was the chosen neural network implementation, as there was an implementation for it in the Python libraries and it fits this kind of data set, being a feedforward network.
Number of iterations was set to 3000, as this was determined to be enough for the algorithm to converge with the default value for tolerance for optimization. The layer size was 100 neurons for the only hidden layer. The activation function used was ReLU, because it is generally a good first choice for feedforward networks. Before MLP was applied, the same standardization as the one applied to the $k$-means training set was applied to the training and test sets.

Because the number of parameters was high (eleven), it was decided that feature selection should be tried to see if the number of features could be reduced. High dimensionality made manual analysis through visualising difficult and also increases computational complexity. Feature selection would also provide insight into which features had the most impact on the CPU loads. Sequential feature selection was used, as it is a simple and relatively efficient method for the selection of features and there are plenty of available resources for its implementation. A sequential feature selection implementation from MLxtend was used in this work.

Sequential feature selection has four different variants: sequential forward selection (SFS), sequential backward selection (SBS), sequential forward floating selection (SFFS) and sequential backward floating selection (SBFS). MLxtend provides implementations of all versions, and for confirmation all four versions were applied to the training set with random forest. MLxtend takes algorithms and methods from scikit learn as the working implementations used in the analysis, and only provides the sequential feature selection solution. Mean absolute error was used as the evaluation criterion. Mean squared error and R$^2$ scores were also considered, but in pre-testing MSE provided wildly differing values and R$^2$ score in MLxtend is un-adjusted, which provides overly optimistic results. For each model, ten-fold cross validation was used at each step. Each method output its choice for the best feature set, and the most chosen feature set between methods was then used to create a regression model.

This model was then evaluated on the test set. Because of the computational intensity of neural networks, only SFFS was used with MLP, and the cross-validation was only five-fold. Full feature set was also not evaluated for MLP. MLP implementation also lacks the ability to measure the importance of the features, so only the result of feature selection was available for analysis.

### 5.3.2. Results

The results for the analysis done on the data from test logs are presented below. Results for anomaly detection, clustering and regression for each configuration are presented in their own sections. The results are then discussed afterwards.

**Anomaly detection**

Both isolation forest and one class SVM were unable to correctly classify the failed cases in the training set. In figures 12 and 13, y-axel value zero represents failed cases and one represents successful cases. CPU load is in the x-axel for display purposes. Green samples are classified normal by the algorithm and red are classified as anomalies. As can be seen from the figures, both methods classified most failed cases...
as normal, while classifying normal cases as anomalies. In the case of the isolation forest, out of the 107 failed cases in the training set, eight were classified as anomalies, and the rest as normal. 144 normal cases were classified as anomalies. The confusion matrix of the results in Table 2 clearly shows how poorly isolation forest classified failed cases as anomalies while also giving many false positives and false negatives. Anomaly detection accuracy for isolation forest was 7.4%. One class SVM classified 36 failed cases as anomalies and 267 normal cases as anomalies. The confusion matrix of the one class SVM results in Table 3 shows a similar poor trend. One class SVM also reported many false positives and false negatives and had an anomaly detection accuracy of 33.6%. The two other configurations provided very similar results.

![Figure 12. Results of isolation forest anomaly detection.](image)

<table>
<thead>
<tr>
<th>Predicted anomaly</th>
<th>Predicted normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual anomaly</td>
<td>8</td>
</tr>
<tr>
<td>Actual normal</td>
<td>144</td>
</tr>
</tbody>
</table>
Figure 13. Results of one class SVM anomaly detection

Table 3. Confusion matrix of anomaly detection results using one class SVM

<table>
<thead>
<tr>
<th></th>
<th>Predicted anomaly</th>
<th>Predicted normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual anomaly</td>
<td>36</td>
<td>71</td>
</tr>
<tr>
<td>Actual normal</td>
<td>267</td>
<td>2660</td>
</tr>
</tbody>
</table>

The high dimensionality of the data set made visual analysis of the results difficult. However, quick analysis indicated the way the algorithms determined anomalous cases gave no meaningful insight into performance monitoring. It seems that the algorithms determined cases with extreme parameter values as anomalous, which is not useful in the context of the work. For this data set, anomaly detection with isolation forests or one class SVM is not feasible.

Clustering

The inertias on all three configurations are illustrated in Figure 14. As can be seen, they are very similar and follow an almost identical trend. Effects of the curse of dimensionality can be observed in the high values of the inertia, as it causes Euclidean distances to bloat. High inertia also signals of a high sum of squares error, which means that the clusters are not very well defined. Ultimately eight was chosen as the number of clusters based on the inertias and analysed further.
As was the case with isolation forest and one class SVM, the high dimensionality made analysing the results difficult. Visualising different feature pairs and manually going through the samples in different clusters showed no clusters that were meaningful for the issue at hand.

For the sake of analysis $k$-means was additionally run with the test result (pass/fail) added into the training data. This time $k$-means created a cluster that contained all 107 failed cases. This can be seen from Figure 15, where zero on y-axel represents a failed case and one represents a successful case. All failed cases are assigned to the orange cluster. While interesting, it is questionable if this is useful for performance monitoring. As the result of the test case is already known, there is no need for an algorithm to point it out. Despite this, $k$-means was still considered potentially useful in the future, as more thorough and invested analysis might provide usable results.
Figure 15. Illustration of result of clustering with eight clusters.

Regression results

The results of the regression analysis are presented here. The numerical results are gathered in Table 4, where they are sorted by configuration. R² values represent the average score for 10-fold cross-validation of the training set, while the MSE and MAE values are for the testing set. The best values for each configuration are highlighted with bold text. As can clearly be seen, random forests outperformed multi-layered perceptron significantly. Both error rates are considerably lower for random forest, and the R² score is higher in all cases.

Table 4. Results of the regression analysis for all three configurations

<table>
<thead>
<tr>
<th>Measure</th>
<th>Random forest full set</th>
<th>Random forest reduced set</th>
<th>MLP reduced set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Configuration 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R²</td>
<td>0.891</td>
<td><strong>0.906</strong></td>
<td>0.847</td>
</tr>
<tr>
<td>MSE</td>
<td><strong>43.793</strong></td>
<td>51.707</td>
<td>563.924</td>
</tr>
<tr>
<td>MAE</td>
<td><strong>4.139</strong></td>
<td>4.418</td>
<td>17.115</td>
</tr>
<tr>
<td></td>
<td>Configuration 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R²</td>
<td>0.880</td>
<td><strong>0.902</strong></td>
<td>0.876</td>
</tr>
<tr>
<td>MSE</td>
<td><strong>30.846</strong></td>
<td>33.026</td>
<td>509.458</td>
</tr>
<tr>
<td>MAE</td>
<td><strong>3.511</strong></td>
<td>3.637</td>
<td>16.604</td>
</tr>
<tr>
<td></td>
<td>Configuration 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R²</td>
<td>0.838</td>
<td><strong>0.855</strong></td>
<td>0.710</td>
</tr>
<tr>
<td>MSE</td>
<td><strong>70.137</strong></td>
<td>91.946</td>
<td>990.777</td>
</tr>
<tr>
<td>MAE</td>
<td><strong>4.910</strong></td>
<td>5.315</td>
<td>21.946</td>
</tr>
</tbody>
</table>
The results of the random forest feature importance analysis are presented in bar graphs in Figure 16. The bars represent the importance of each feature relative to all features, and they sum up to one. The red line shows the development of how much each feature adds to the general importance. The importance of features for the sets selected with feature selection were also measured. They generally follow the same trends as the graphs in Figure 16 and can be seen in Appendix 2.

![Figure 16. Bar graphs representing feature importance’s for all three configurations.](image)

Table 5 shows the results of feature selection for each configuration. The results for both random forests and MLP are shown. The four sequential feature selection methods generally all came to the same conclusion about the best feature set for random forests. Only for configuration 1 did SFS disagree with the rest. The majority vote was used to choose the feature set that the other three sequential feature selection variants found best. In hindsight, using all four variants of the sequential feature selection was unnecessary. While it brings confidence in the results, SFFS and SBFS are essentially updated versions of the simpler variants and using them all is redundant.
Table 5. Feature sets chosen by feature selection

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Random forests</th>
<th>Multi-layered perceptron</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x2, x3, x5, x6, x8, x10, x11</td>
<td>x1, x3, x5, x6, x7, x8, x9, x10, x11</td>
</tr>
<tr>
<td>2</td>
<td>x1, x2, x3, x6, x8, x10, x11</td>
<td>x1, x2, x3, x5, x7, x8, x9, x10, x11</td>
</tr>
<tr>
<td>3</td>
<td>x1, x2, x3, x6, x8, x10, x11</td>
<td>x3, x4, x6, x7, x8, x9, x10, x11</td>
</tr>
</tbody>
</table>

5.3.3. Discussion of Results

Configuration 1 and 2 were very similar in performance, with configuration 2 being somewhat better. Configuration 3 was worst on all measures. It should be noted that the configurations have their loads on somewhat different scales, due to the different feature representations that also affected the way loads were calculated. This can be seen from Table 6, where the mean values of the training set and the test set for each configuration is presented. This difference in scale partially explains the difference in error rates and taking it into account, configurations 1 and 2 can be considered essentially equal in performance. While some of the difference in error rates can be explained by the bigger scale of CPU loads that configuration 3 has, the difference is too big to result only from that. Also, the R² scores are significantly worse. It can be confidently said that configuration 3 performed the worst.

Table 6. Mean values of CPU loads for the three configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Training set mean CPU load</th>
<th>Testing set mean CPU load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration 1</td>
<td>167.2</td>
<td>160.6</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>133</td>
<td>126.9</td>
</tr>
<tr>
<td>Configuration 2</td>
<td>181.7</td>
<td>176.3</td>
</tr>
</tbody>
</table>

Overall random forest showed promising results on all configurations. Mean error values were quite low and the mean R² value of cross-validation on training sets was close to 90% on all configurations. It should be noted that the cross-validation R² scores had high variance for all variations, ranging from 0.70 to high 0.90. This signals that the model’s performance is reliant on the part of the training set that is used, and more data is needed to even out this discrepancy. It should also be noted that un-adjusted R² score was used as the performance measure in cross-validation, which gives a somewhat optimistic result. However, the adjusted R² score of the test set showed that the difference is not significant. Figure 17 showcases the plot of measured and predicted CPU loads for the test set with configuration 1. As can be seen, they line up in linear fashion, although some outliers can be seen. Considering the quite unorganized test setup with little optimization of model parameters, these are excellent results and signal that the CPU loads are dependent on at least some of the L2 test parameters.
Multi-layered perceptron performed distinctly worse than random forest. It can be clearly seen from the error values that MLP had greater mean errors, and Figure 18 clearly shows that MLP shows greater variance between the predicted and measured values than the random forest equivalent. The figure is for configuration 1 with a reduced feature set; other configurations show similar results. MLP was also computationally more demanding. Running only SFFS on MLP with five-fold cross-validation took longer than running all four sequential feature selection variants with random forest with ten-fold cross-validations each. As the planned monitoring tool is meant to become part of the testing workflow in production, computational heaviness and long run times are not acceptable. The weaker results can be explained with a lack of fine-tuning of the neural network and could be improved by tweaking the parameters on said neural network. The network had only one hidden layer with 100 neurons and was run with almost entirely default values, which clearly do not fit the task at hand. This is supported by the fact that when another layer was added to the network and the transfer function was changed to a hyperbolic tangent function later, the performance improved significantly, although not enough to match random forest. While the neural network implementation allows for extensive fine tuning of the training and network parameters, the time and effort to optimize the neural network are not worth it when random forest already provides better results with little fine tuning.

Figure 17. Measured and predicted loads of test set for configuration 1 with random forest. Reduced feature set was used.
The feature selection and the analysis of the relative importance of features yielded interesting results. For performance, feature selection did not provide meaningful improvements. While the $R^2$ improved, the error rates rose. The biggest result was the clear importance of feature $x_3$ on the CPU loads. It has the highest importance in every comparison, by a wide margin in all but one case. Another interesting insight was how features $x_4$, $x_7$ and $x_9$ are always removed in feature selection for random forests, despite their relatively high importance in the full set. This can be explained by correlation. Figure 19 shows the visualised correlation matrix of the features in configuration 1. Pearson’s correlation coefficients were used. As can be seen, feature $x_4$ correlates heavily with $x_3$, with a coefficient value of 0.82. Similarly, $x_7$ and $x_9$ correlate heavily with $x_6$ and $x_8$, with values 0.84 and 0.56 respectively. As was discussed in the feature importance part of the literature review, correlation between features causes bias in random forest feature importance calculations. In this case it causes features $x_4$, $x_7$, and $x_9$ to disappear, despite their relative high importance in the full feature set. Feature $x_5$ also correlates significantly with $x_3$ and $x_4$, and it is missing from the features chosen for configurations 2 and 3. The correlation matrices of configurations 1 and 2 are nearly identical. Configuration 3 had some differences, notably $x_5$ no longer correlated as heavily with $x_3$ and $x_4$, but the other correlations presented previously were still present.
Figure 19. Correlation matrix of features in configuration 1.

Figure 20 shows a comparison of the error rate development for a different number of features in the feature selection. The error values are negative as the scikit-learn library uses negative error values to better visualize the development of the error rate by having the line rise when a better error value is achieved. The left plot is for random forests and the right one is for MLP. These plots are for configuration 1, and the other configurations showed similar development. All error rate development plots in explorative analysis can be seen in Appendix 3. From Table 5 and Figure 20 it can be seen that the peak performance is found around seven features for random forests and nine features for MLP, after which adding features generally slightly decreases performance. The figure also shows that the most important features are reached around five to six features, after which the error rate plateaus and only minor changes happen. This is in line with the previous feature importance graphs, which show a similar trend.

It should be noted that the MLxtend SFFS solution provided suspicious error rates. For example, peak MAE for the best feature set for configuration 1 in the left figure is 19.5007, but the same metric calculated independently for the best feature set is 1.2665. The same difference was seen for other configurations. This is despite MLxtend using the same scikit-learn regressors and methods and the same training set in its calculations that were used to calculate the values in Table 4. This disparity is not explained in the documentation of the MLxtend API.
Figure 20. Error rate development with feature selection. Random forest on the left, multi-layered perceptron on the right.

The results of the explorative analysis were discussed within the company. The third configuration was first discarded because of its lowest performance and the extreme values that few cases introduced into the model. This can be seen from Figure 21, where some cases are shown reaching absurd loads. These were considered not representative of the system. The good results of configurations 1 and 2 were noted, but there were still doubts whether they properly represented the system.

Ultimately all three configurations were deemed wrong in some way. Configurations 1 and 2 were the closest to a true representation of features regarding the test cases, but even they skewed the representation for some cases. A singular representation completely modelling all test cases would be significantly more complicated and require a lot more parameters. Another option would be to have multiple different representations to account for different test variants. Both options are too complicated, both for this thesis and for the intended test result monitoring.

However, the good results were found encouraging and the findings about feature importance meaningful. To make a compromise with the feature representation, a simplified version of one of the configurations was chosen with some parameters being set on a certain level and this was determined to be the ground truth for this work. This was deemed to be in line with most cases being tested in the environment, and a good enough representation for the rest to allow for performance monitoring in the scope of this work. It was decided that the regression model would become the goal of this work. Researching the effects of L2 parameters on CPU loads and which parameters affected the loads most became the focus of this work and the research question to be answered.

Neural networks were discarded for this work. They require a lot of parameter fine tuning and expert level knowledge for effective use, which are beyond the scope of this work. Additionally, even the small neural network with only one hidden layer ended up being very computationally demanding, signalling that a proper neural network could become a bottleneck if introduced into production. The results for MLP were still noted as promising for possible future work, as a better optimized network could provide benefits in some other context.
Figure 21. Comparison of measured and predicted loads for configuration 3. Notice the extreme CPU loads that some instances have.
6. IMPLEMENTATION AND EVALUATION

The ultimate goal of this work was to build a model capable of estimating CPU loads from the L2 test parameters and estimating which parameters affect the loads the most. The results of the explorative analysis led to regression with random forest being the method for achieving this goal. To this end, the script for parsing logs was updated to match the new parameter representation. Data gathering continued, with result logs being pulled and stored each day. When enough data was gathered the work on training and optimizing the random forest regression models started.

6.1. Test Parameters and Data Set

Due to the change in feature representation, the input vector has changed. As \( x_1 \) of the original feature set was set as a constant for the ground truth, it was discarded from the input vector. The new number of input parameters is ten, and the new input vector \( X \) for a sample size of \( n \) and feature amount \( k=10 \) is similar to the vector presented in (12). This means that \( x_1 \) of the new feature set is equivalent to \( x_2 \) of the original set, with other features following the same pattern. The output vector \( Y \) sized \( (n \times 1) \) remains unchanged from that of (12).

Additionally, due to further study into the hardware, the CPU loads of which are estimated here, it was decided that one hardware configuration being tested required the CPU loads to be split in two. For this reason, another more focused model was also created to take this difference to account. This model will be referred to as model 2, and the other model as model 1. This model has nine input parameters, as \( x_{10} \) becomes a constant. Features \( x_1 \) to \( x_9 \) are the same as those of model 1. Input vector \( X \) for this model for a sample size of \( n \) and feature amount \( k=9 \) is presented at (13). As the output is now split in two, the output vector \( Y \) is now sized \( (n \times 2) \) and is also shown in (13).

\[
X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ x_{31} & x_{32} & \cdots & x_{3k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix}, \quad Y = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \\ y_{31} & y_{32} \\ \vdots & \vdots \\ y_{n1} & y_{n2} \end{bmatrix} \tag{13}
\]

The data gathered for analysis was split into the development set and the validation set. The development set is used for model training with the training set and model selection with the testing set. The training and test sets are taken from the development set using random sampling. Random sampling can be used, as the jobs are always run as clean runs, so each run is independent and does not have any dependence on the runs before it. The validation is a separate set of samples taken from the end of data gathering period, to validate that the model generalizes adequately. The split is approximately 70-15-15 for the training set, the testing set, and the validation set,
respectively. 60-20-20 split was also considered, but a bigger training set was preferred especially for the smaller sample size of model 2, and the testing and validation sets remained large enough.

The data was cleaned before it was used. Duplicate samples and samples that had missing values or otherwise clearly anomalous values were dropped. Additionally, standardized residuals were calculated for the prediction errors. $E$ is vector of errors, that is calculated in (15) by subtracting predicted values in vector $\hat{Y}$ from the vector $Y$ of actual values. The standardized residuals were calculated with formula (16), where $S$ is the vector of standardized residuals of errors and $\text{std}$ refers to the standard deviation.

Any sample whose standardized residual was larger than 2 was investigated as a possible outlier. This procedure helped discover more anomalous samples and additionally some specialized jobs that had to be dropped from the analysis as it was noticed that they produced vastly different CPU loads compared to jobs with similar parameter values. This discrepancy was caused by factors that could not be explained with features used in this analysis, and thus their inclusion in the training and testing data would only weaken the model.

\begin{align*}
E &= Y - \hat{Y} \\
S &= \frac{E - \text{mean}(E)}{\text{std}(E)} \\
\end{align*} 

The size of training set for model 1 was ultimately 7150 samples, while the testing set was 1570 samples. A sample represents a single run of a single test case. The validation set was 1597 samples. For model 2 the sizes are 2112 samples for the training set, 451 samples for the testing set and 443 samples for the validation set. The model 1 data set includes 127 unique jobs with 417 unique test cases and the model 2 data set includes 39 unique jobs with 123 unique test cases.

6.2. Model Selection

Model selection in machine learning is not always only about choosing between different methods and algorithms. When creating machine learning models, the parameters that are used to predict new samples are learned from the training data. There are also parameters that are set before any training takes place that affect both the training and the quality of the resulting model. These are called hyperparameters, and they are algorithm dependent. Tuning these parameters is important for optimizing the model, as the best set of hyperparameters are generally case dependent.

6.2.1. Random Forest Regressor

As was decided after the explorative analysis, random forest was chosen as the algorithm for regression. This choice was dictated by the good performance during the explorative analysis, the ability to inherently measure the importance of features and
the availability of a ready implementation with scikit learn. Additionally, the Python implementation of random forest can handle multi-output situations without extra effort, which is essential for the second model.

Gradient boosting trees were also considered due to their similarity to random forests and potentially better performance to random forests in some cases [5 p. 590-592]. However, quick testing with similar sized trees showed similar or worse results to random forest. Additionally, the gradient boosting regression implementation in scikit-learn library does not support multi-output cases without adding extra modules to wrap the regressor. Gradient boosting trees were quickly discarded, as the extra work optimizing them in order to possibly outperform random forest was not worth it.

As a reminder, random forests are ensembles of decision trees created through bagging, where the output of the model in regression is the average of the outputs of each tree. What makes them unique is the random sampling of both samples and features for trees in the forest, which helps de-correlate the trees and reduce overfitting.

The random forest implementation used in this work is from the scikit-learn Python library\(^3\). The implementation is based on the CART method explained in the literature review. It is a full random forest regressor implementation with an inbuilt feature importance estimator and the ability to easily tune the various hyperparameters. For random forests, the hyperparameters are related to the number of trees, the size and architecture of the trees, and the way the trees are built.

Two kinds of bias affecting random forest feature importance estimation was discussed in the literature review. The categorical bias favoring features with more categories is unlikely to be a problem in this work, as the two categorical features have the same number of features for model 1 and the other categorical feature is a constant and not present for model 2. However, the bias caused by correlation between features is relevant to this work, as was shown by the explorative analysis. When examining the results of the feature importance analysis, this needs to be considered.

### 6.2.2. Hyperparameter Selection

In order to get the best performance out of the random forest model, the hyperparameters require tuning. Expert knowledge and experience can be used to predict a good set of hyperparameters, that usually perform well. Python implementations of different algorithms generally also have well balanced default values for hyperparameters, which provide a good starting point for analysis. However, finding the optimal set of hyperparameters for a particular case requires trying out different sets of parameters and comparing their performance to find the best set.

---

Hyperparameter tuning is one of the best ways to improve model performance without changing the algorithm or gathering more data and feature engineering.

The scikit-learn implementation of random forest exposes multiple essential hyperparameters for tuning. Probably the most important is the number of trees. More trees generally mean better performance, although the performance gains generally stagnate at some point. More trees also increase the computational load of the algorithm, so finding the balance between performance and load is important. Another important hyperparameter is the maximum number of features considered when looking for the best split at nodes. Having more features decide the split strengthens individual trees in the forest but lowering the number of features reduces correlation between trees. This reduction in correlation can offset the negative effect that weaker trees have on generalization error [26].

Other useful tunable hyperparameters include maximum tree depth, the minimum number of samples placed in a node before it is split and the minimum number of samples in a leaf node. The default values for these parameters in the scikit-learn implementation lead to fully grown and unpruned trees, which can easily lead to very large trees with large data sets. Fully growing the trees is often unnecessary and consumes memory, which is why the parameters should be tuned, in addition to their effect on predictor performance. The last hyperparameter to tune is whether to use bootstrapping which controls whether the random sampling happens with replacement or not.

Table 7 lists the hyperparameters that were tuned and their candidate values. The candidate values were chosen with expert knowledge and various tutorials\textsuperscript{4}. The ‘auto’ value for the number of features uses all available features, while ‘sqrt’ uses the square root of the number of all features, which for both ten and nine features rounds to three. Seven features are considered as a median value between the other two. The None value for maximum tree depth leads to fully grown trees.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Hyperparameter} & \textbf{Candidate values} \\
\hline
Number of trees & 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000 \\
\hline
Number of features & ‘auto’, ‘sqrt’, 7 \\
\hline
Max tree depth & 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None \\
\hline
Min samples per split & 2, 5, 10 \\
\hline
Min samples in leaf nodes & 1, 2, 4, 6, 8 \\
\hline
Bootstrap & True, False \\
\hline
\end{tabular}
\caption{Initial tuned hyperparameters and their candidate values}
\end{table}

\textsuperscript{4} This tutorial by William Koehrsen inspired both the initial candidate values and the whole hyperparameter search process \url{https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74}
Trying out different sets of hyperparameters quickly becomes very time-consuming as the number of parameters and the range of their values increases. For the mentioned set of different hyperparameters and their values a total of $10^3 \times 3 \times 11^3 \times 3^5 \times 2 = 9900$ iterations would be needed. Fortunately, scikit-learn offers methods to both automate and randomize the search for hyperparameters.

The general framework of the optimal hyperparameter estimation goes as follows. First, a random forest regressor with the default hyperparameter values is built and fitted with the training data and evaluated with the testing data in order to create a baseline performance and to see whether hyperparameter optimization actually improved the model performance. The default random forest regressor parameters are listed in Table 8. Then, the different sets of candidate values in Table 7 are tested and evaluated to find the parameter set that produces the best performance.

Table 8. Default hyperparameter values of random forest regressor

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Default values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of trees</td>
<td>10</td>
</tr>
<tr>
<td>Number of features</td>
<td>‘auto’</td>
</tr>
<tr>
<td>Max tree depth</td>
<td>None</td>
</tr>
<tr>
<td>Min samples per split</td>
<td>2</td>
</tr>
<tr>
<td>Min samples in leaf nodes</td>
<td>1</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>True</td>
</tr>
</tbody>
</table>

An exhaustive search of all combinations would be unfeasible, so a randomized search with a fixed number of iterations is conducted instead to reduce the search range. Scikit-learn offers a function called RandomizedSearchCV\(^5\) that can be used to try out a fixed number of randomized parameter sets and to choose the best performing one. It also has an option to do cross-validation for each iteration. MSE is used as the performance metric in cross-validation to find the best parameter set. This was different from the explorative analysis, where MAE was used as the performance metric. The change was made, as MSE better reacts to larger errors than MAE, and reducing large errors is beneficial for this work. In this work 3000 iterations with three-fold cross-validation was performed. This leads to 9000 total runs, which is acceptable as it cuts down the amount of iterations from an exhaustive search by almost 70 percent, while still going through a meaningful chunk of variations. With three-fold cross validation, the exhaustive search would require 29700 runs.

---

The best parameter set found by randomized search is then evaluated with the test set. The performance metrics calculated are MAE, MSE and adjusted R². If the results do not improve from the baseline performance, the randomized search is run again with a different random seed number. If the performance does improve, a new, more focused set of hyperparameter candidate values is derived from the best parameter set.

These new values are then tried with the GridSearchCV function, which does an exhaustive search of all parameter combinations of the given candidates. The exhaustive search is also run with three-fold cross-validation. The best parameter set found by the exhaustive search is again evaluated with the test set, and performance compared to both the baseline performance and the performance of the best set of randomized search. If the performance is deemed good enough, the search stops there, and that parameter set is chosen for the final model, but if there seems to be room for improvement, the random search is performed again with a different random seed number and possibly a different range of parameters if improvement cannot be found. It should be noted that CPU loads generally fluctuate between runs a little even within the same revision with identical parameter values, which introduces irreducible error into the model. The model will never reach perfect predictive accuracy and knowing when to stop the optimization is necessary.

**Model 1**

The performance of all three versions of the regressor are listed in Table 9. Base regressor refers to the baseline regressor with default hyperparameters, best random refers to the best parameter set found by the randomized search, and best exhaustive refers to the best parameter set found by the narrowed down search. Table 10 lists the best parameter set found by the random search and the exhaustive search. The parameter set found by the exhaustive search is also the hyperparameter set that will be used in the final model. Table 11 shows the new parameter candidates for the exhaustive search based on the parameters found by the random search.

Ultimately, the hyperparameter optimization decreased the mean squared error by 27% and the absolute mean error by 5%. Adjusted R² increased by 0.11%. The performance of the model on the test set is good and there is no need for further search. Any significant improvement is unlikely, and as the random search took almost five hours to complete, it is not meaningful to run it again. The best hyperparameter set of model 2 was also tried but brought no improvement in performance.

---

Table 9. Table of results for different sets of hyperparameters for model 1

<table>
<thead>
<tr>
<th>Measure</th>
<th>Base regressor</th>
<th>Best random</th>
<th>Best exhaustive</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>2.869</td>
<td>2.7511</td>
<td>2.7251</td>
</tr>
<tr>
<td>MSE</td>
<td>33.4349</td>
<td>24.2756</td>
<td>24.2720</td>
</tr>
<tr>
<td>Adj. $R^2$</td>
<td>0.9961</td>
<td>0.9972</td>
<td><strong>0.9972</strong></td>
</tr>
</tbody>
</table>

Table 10. Best parameters found by random search and exhaustive search

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Random set</th>
<th>Exhaustive set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of trees</td>
<td>1400</td>
<td>1300</td>
</tr>
<tr>
<td>Number of features</td>
<td>‘sqrt’</td>
<td>4</td>
</tr>
<tr>
<td>Max tree depth</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Min samples per split</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Min samples in leaf nodes</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 11. New focused candidate values for exhaustive search

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Candidate values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of trees</td>
<td>1200, 1300, 1400, 1500, 1600</td>
</tr>
<tr>
<td>Number of features</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>Max tree depth</td>
<td>10, 20, 30, 40, 50</td>
</tr>
<tr>
<td>Min samples per split</td>
<td>9, 10, 11</td>
</tr>
<tr>
<td>Min samples in leaf nodes</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>False</td>
</tr>
</tbody>
</table>

Model 2

The performance of all three versions of the regressor are listed in Table 12. Base regressor refers to the baseline regressor with default hyperparameters, best random refers to the best parameter set found by the randomized search, and best exhaustive refers to the best parameter set found by the narrowed down search. As model 2 has two values in its output, both the raw values for each output and the uniform average are show. Only the uniform average is used for performance comparison, however.

Table 13 shows the best parameter sets found by the random search and by the exhaustive search, which is also the parameter set chosen for the final model. Table 14 shows the candidate parameter values for the exhaustive search based on the best parameter set of the random search. Ultimately, hyperparameter optimization improved the model performance by 55% for MSE, 15% for MAE and 0.12% for adjusted $R^2$. The performance of the model on the test set is good and there is no need for further optimization. Any significant improvement is unlikely. The best hyperparameter set of model 1 was also tried but brought no improvement in performance.
Table 12. The results for all three versions of the regressor

<table>
<thead>
<tr>
<th>Measure</th>
<th>Base regressor</th>
<th>Best random</th>
<th>Best exhaustive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$y_1$</td>
<td>$y_1$</td>
<td>$y_1$</td>
</tr>
<tr>
<td>MAE</td>
<td>2.5179</td>
<td><strong>2.1802</strong></td>
<td>2.1812</td>
</tr>
<tr>
<td></td>
<td>Un.Avg. 1.6919</td>
<td>1.4447</td>
<td><strong>1.4438</strong></td>
</tr>
<tr>
<td>MSE</td>
<td>22.7229</td>
<td>10.7688</td>
<td><strong>10.7408</strong></td>
</tr>
<tr>
<td></td>
<td>Un.Avg. 13.3624</td>
<td>6.0014</td>
<td><strong>5.9388</strong></td>
</tr>
<tr>
<td>Adj. $R^2$</td>
<td>Raw 0.9979</td>
<td>0.9994</td>
<td><strong>0.9999</strong></td>
</tr>
<tr>
<td></td>
<td>Un.Avg. 0.9980</td>
<td>0.9992</td>
<td><strong>0.9992</strong></td>
</tr>
</tbody>
</table>

Table 13. Best parameters found by random and exhaustive search

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Random set</th>
<th>Exhaustive set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of trees</td>
<td>2000</td>
<td>2100</td>
</tr>
<tr>
<td>Number of features</td>
<td>‘sqrt’</td>
<td>2</td>
</tr>
<tr>
<td>Max tree depth</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>Min samples per split</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>Min samples in leaf nodes</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 14. New focused candidate values for exhaustive search

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Candidate values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of trees</td>
<td>1800, 1900, 2000, 2100, 2200</td>
</tr>
<tr>
<td>Number of features</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>Max tree depth</td>
<td>30, 40, 50, 60, 70</td>
</tr>
<tr>
<td>Min samples per split</td>
<td>9, 10, 11</td>
</tr>
<tr>
<td>Min samples in leaf nodes</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>False</td>
</tr>
</tbody>
</table>

6.3. Model Validation and Importance of Features

After the hyperparameters were selected and the final model configurations decided, the models were finally evaluated with a separate validation data set. At a certain point it was deemed that there was enough data for a proper development set, and all data gathered after that became the validation set. This way the validation set would be as independent as possible from the data used to train the model, and would also be the most up to date, thus reflecting the current state of the testing environment.

Both models were validated with their respective validation sets. For model 1, the validation set is sized 1597 samples, and the same for model 2 is 443 samples. MAE, MSE and adjusted $R^2$ were used to evaluate the performance of each model. For model 2, both uniform average and raw values of both target values are presented. The importance’s of features were analyzed with the in-built feature importance estimator. SFFS was also run with the training set as an additional measure of feature importance,
and to see if feature reduction would further improve performance. This was done despite the odd error rates that the MLxtend was found to provide during the explorative analysis, to see if the problem persisted.

### 6.3.1. Model 1

The performance of model 1 on the validation set is shown in Table 15. For comparison, the same metrics for the testing set are also shown. As can be seen, the validation set had significantly worse performance than the test set. Visualizing the relationship between measured and predicted in Figure 22 shows some clear outliers, which affected the error rates significantly. Using the same method of standardized residuals as was outlined previously to isolate the outliers showed that the they were mostly anomalies, with a couple of wrong predictions of normal results. Removing fourteen of the most blatant outliers with standard residual over four from the validation set improved the performance of the model considerably, as can be seen from the table. Especially MSE lowered considerably, which makes sense as MSE is sensitive to larger errors. Further performance gain could possibly be gained with removal of more outliers but is not necessary to prove the good performance of the model. Overall, the performance of the model is acceptable.

The best performing feature set chosen by SFFS is shown in Table 16. The development of the error rate is shown in Figure 23, where the error rate plateaus after six features. This is in line with the feature importance graph in Figure 25, where the features after the fifth important feature have very similar, low importance. The best set included eight features, removing $x_1$ and $x_4$. While $x_1$ is understandable, as it is shown to have low importance, $x_4$ is more surprising, as it is the fourth most important feature according to the feature importance estimation. It should be noted that both removed features were somewhat correlated with multiple other features, which might have affected the outcome.

The results of using this new feature set to train the model and evaluating it with the cleaned validation set are shown in Table 15. While small performance gain is achieved with MSE, the relative importance of one of the removed features and the possible bias caused by correlating features makes it so that the improvement is not major enough to warrant the reduction of the number of features. Also, the same discrepancy in error rates that was seen during explorative analysis was seen again, which casts heavy doubt on the validity of the whole SFFS implementation. MSE peaked at 30.944 for the cross-validation that was used to compare performance in SFFS implementation, yet calculating the same metric for the training set with the eight chosen features gives 7.5453.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Test</th>
<th>Validation</th>
<th>Cleaned</th>
<th>SFFS Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>2.7251</td>
<td>3.7085</td>
<td>3.285</td>
<td>3.2935</td>
</tr>
<tr>
<td>MSE</td>
<td>24.2720</td>
<td>50.8294</td>
<td>25.0466</td>
<td>24.7843</td>
</tr>
<tr>
<td>Adj. R²</td>
<td>0.9972</td>
<td>0.9941</td>
<td>0.9971</td>
<td>0.9971</td>
</tr>
</tbody>
</table>
Figure 22. Comparison of predicted and measured loads in full uncleaned validation set for model 1.

Table 16. Best feature set according to SFFS

<table>
<thead>
<tr>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₂, X₃, X₅, X₆, X₇, X₈, X₉, X₁₀</td>
</tr>
</tbody>
</table>

Figure 23. Error rate development for model 1.
To evaluate correlations between features, a correlation matrix of the training set was calculated. The matrix is shown in Figure 24. As can be seen, features $x_2$ and $x_3$ correlate heavily with each other, as do $x_5$ and $x_6$. Both $x_1$ and $x_4$ correlate to a lesser extent with multiple other features. The importances of features estimated based on their effect on the gini impurity are shown in Figure 25. As can be seen, feature $x_2$ dominates by accounting for almost 40% of the impurity reduction. The three most important features account for almost 70% of the impurity reduction. Despite their heavy correlation with each other, $x_2$ and $x_3$ are the two most important features.

Figure 24. Correlation matrix of features for model 1.
Figure 25. Feature importances in model 1.

6.3.2. Model 2

The performance of model 2 on the validation set is shown in Table 17. For comparison, the same metrics for the testing set are also shown. As can be seen, the validation set performed similarly but slightly worse than the testing set, except for having better MAE for $y_1$. The visualization seen in Figure 26 also shows that there are no major outliers for either output target. The cleaning and re-evaluation done for the validation set of model 1 is not necessary as there are no clear outliers.

The best feature set found by feature selection is shown in Table 18. The set has seven features, with $x_5$ and $x_6$ being dropped from the set. The development of error rate is shown in Figure 27, which shows that the error rate plateaus after three or four features. This is in line with the feature importance graph in Figure 29, where the four most important features account for almost 80% of the impurity reduction. The removed features do not have very high importance, although $x_5$ does still account for approximately 10% of the impurity reduction. The reduced feature set provides no performance gain, so removing any features from the data set is not meaningful based on this evaluation.

Again, the discrepancy in error rates that was seen during explorative analysis and model 1 evaluation was present, which further questions the validity of the SFFS implementation. MSE peaked at 6.753 for the cross-validation that was used to compare performance in SFFS implementation, yet calculating the same metric for the training set with the seven chosen features gives 2.0540.
Table 17. Performance of model 2 on various data sets

<table>
<thead>
<tr>
<th>Measure</th>
<th>Test</th>
<th>Validation</th>
<th>SFFS validation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y₁</td>
<td>y₂</td>
<td>y₁</td>
</tr>
<tr>
<td>MAE</td>
<td>Raw</td>
<td>2.1812</td>
<td>0.7065</td>
</tr>
<tr>
<td></td>
<td>Un.Avg.</td>
<td>1.4438</td>
<td>1.4461</td>
</tr>
<tr>
<td>MSE</td>
<td>Raw</td>
<td>10.7408</td>
<td>1.1369</td>
</tr>
<tr>
<td></td>
<td>Un.Avg.</td>
<td>5.9388</td>
<td>7.0280</td>
</tr>
<tr>
<td>Adj. R²</td>
<td>Raw</td>
<td>0.999</td>
<td>0.9995</td>
</tr>
<tr>
<td></td>
<td>Un.Avg.</td>
<td>0.9992</td>
<td>0.9991</td>
</tr>
</tbody>
</table>

Figure 26. Comparison of predicted and measured loads for $y₁$ (left) and $y₂$ (right) in full validation set for model 2.

Table 18. Best feature set according to SFFS

<table>
<thead>
<tr>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₁, x₂, x₃, x₄, x₇, x₈, x₉</td>
</tr>
</tbody>
</table>

Figure 27. Error rate development for model 2.
To evaluate correlations between features, a correlation matrix of the training set was calculated. The matrix is shown in Figure 28. Just like in model 1, features $x_2$ and $x_3$ correlate heavily with each other. $X_5$ and $x_6$ also correlate, but to a lesser extent than with model 1. There are other differences in correlations as well, as $x_1$ is now less correlated with other features, while correlations now arise among other features. $x_3$ and $x_4$ correlate heavily, as does $x_3$ with both $x_6$ and $x_7$. In general, the features seem to correlate more with each other than in model 1. This can be also seen in the feature importance graph for model 2 shown in Figure 29. The graph is not as steep as the equivalent for model 1. The correlation bias most likely evens out the importances of the features. Even so, the same four features are still the most important, with $x_2$ and $x_3$ being the most important.

![Figure 28. Correlation matrix of data set for model 2.](image)
Both models performed very well on the validation set, signaling that they model the relationship between the L2 parameters and CPU loads appropriately. Model 2 performed better than model 1, which is to be expected as it is a more focused model. The samples are more homogenous, which could also explain why the features are more correlated with each other. Model 1 has more variance in its features, as it covers multiple different hardware configurations compared to the one of model 2. Both models also performed significantly better than the ones built during the explorative analysis, showing the positive effect that more data and hyperparameter optimization can have on performance. The simplified feature representation very likely also had an effect.

Despite the correlation biases, it can be safely said that features $x_2$ and $x_3$ are the most influential features in general. They have appeared as the most important features for all variations of the data sets and feature sets used in this work for regression, excluding the reduced feature sets for explorative analysis where $x_3$ was removed. The validity of those feature sets is highly questionable due to the implementation used. $X_4$ and $x_7$ are the next most influential for both models.

One thing that could additionally explain the weaker performance of model 1 is that concept drift happened in some test cases during the data gathering period. This is showcased in Figure 30, where the CPU loads of one test case during the period are shown. As can be seen, the trend changes significantly around midway of the period. This came up during the data cleaning, as model 1 could not correctly predict the loads for these cases. This further highlights the fact that while the models can generally predict loads accurately based on the used L2 test parameters, there are still additional factors that affect the CPU loads.
The predictive models built during this work are employed for performance monitoring of capacity test jobs in the company’s testing environment. It is at the core of a tool that daily outputs a report on test case performance based on the results. The tool is meant to guide efforts to find problems and performance issues by analyzing the results of test runs using the two predictive regression models.

The high-level workflow of the tool is outlined in Figure 31. First, the tool gathers the test logs from relevant jobs by pulling them from Jenkins. Then, the test logs are parsed with the log parser implemented during this work to encode the relevant data into .csv files. After this, the data is input into the decision function, where it is evaluated with the built models. Any result that does not fall within the acceptable limits of the models is considered potentially problematic and highlighted for the report. Lastly, a report is composed and published in Jenkins for the developers to read.
The decision function uses the built models to evaluate the results. The models have to be built with the hyperparameters defined in this work and trained with the data gathered during and after this work each time the tool is run. This is because while the models built with the scikit-learn library can technically be made persistent and stored in non-volatile memory, they also come with maintainability and security issues. The models’ internal representation may change when storing and reloading them, and reloading models built on a different version of scikit-learn is not supported. Rebuilding and retraining the models is not a problem, as the tool does not have real time requirements and retraining the models does not take very long.

The decision function predicts the loads of the new result with the models based on their parameters. The predicted loads are then evaluated using standardized residuals of the errors of the predictions. Any samples whose standardized residual is over four is highlighted as a possible problem case. The mean and standard deviation values used for calculating standardized residuals are not calculated from the errors of the new results but are taken from the values calculated during the validation procedure. This way they better represent the general mean and standard deviation of errors and not those of the current errors. The mean and standard deviation values and the threshold value for suspicious results act as tunable parameters for the tool, that can be changed when needed.

To keep the models up to date the new data is then added to the training data that is used to train the models next time. The outliers are removed before this. After the decision function has run and results are ready, a report is composed and published in
the tool’s job page in Jenkins. The report names the jobs and test cases that could be problematic based on the evaluation of the results.
The goal of this thesis was to explore the possibilities of utilizing machine learning for automated performance monitoring in the target system component testing environment. Although multiple different approaches were tried, eventually regression of CPU loads with random forests was chosen as the focus. CPU loads of the system under testing are one of the criterions used to evaluate the performance of the system. Building predictive models capable of estimating the loads from Layer 2 parameters can be used to determine possibly problematic test results by isolating results that cannot be correctly predicted by the model.

Two models were built using real test result data from the testing environment. The models were validated with real data that was gathered separately from the training samples. The validation showed that both models perform very well, with low error rates and the independent variables explaining almost all the variation in CPU loads. In the scope of this work the goal can be considered met.

While the results are excellent, some matters should be considered. The data used is skewed towards certain values of the features. This is the result of using real data, as a lot of the test cases in the environment are quite similar with minor variations of the features. The models are likely at least somewhat overfit to these kinds of test cases. This is not necessarily a problem, as long as the models still apply to the target system. Another concern is the method in which these models are utilized for performance monitoring. Using standardized residuals to detect problematic results is very reliant on the correct choice of the threshold parameter and the mean and standard deviation values. The tool will very likely require close observation and fine tuning of these parameters when it is deployed.

The feature representation used in this work is a compromise made for the sake of simplifying the work and the models. The representation is likely correct for the majority of the test cases but is questionable for some of the more complex cases. According to the evaluation the feature representation models the CPU loads well and is enough for the scope for this work. Still, the feature representation might have to be revisited if problems emerge. Additionally, there are additional L2 parameters that were not considered for this work. As was mentioned during the analysis, some test cases had to be dropped from the training samples as the models could not explain their results. It is likely that there are still factors affecting CPU loads and finding them is probably grounds for future research.

As was mentioned during test parameter definition, another work done in the same company was used as the basis for this work. Kananen used statistical analysis and multiple linear regression to estimate CPU loads in the Cloud Radio Access Network (C-RAN) architecture [59]. While there are differences between the parameters used and Kananen focused on a single architecture compared to the more general approach of this work, the goals are relatively same, and the results are comparable. Kananen’ s model could explain 81.4% of the variance in the CPU loads, while the random forest models in this work could explain 99% of the variance. This is a clear improvement, especially considering the wider range of hardware covered by this work and the simpler cases used in Kananen’ s work. The better performance can be explained by multiple factors.
The number of samples used in this work is vastly larger than that of Kananen’s, which gives the models more to work with. The use of machine learning also gives an edge over manual statistical analysis, as it allows the machine to do the modeling, which is generally better at finding the relationships between dependent and independent variables. Kananen also found that his assumption of linearity between the independent and dependent variables might not be true. As random forest is a non-linear, non-parametric algorithm, it is natural that it would perform better if the relationship is non-linear. Linearity of the relationship was not investigated in this work, although random forest performing better than linear regression would support a non-linear relationship. Kananen implemented a tool to estimate CPU loads based on the parameters to help decision making for developers when implementing test cases. The models built in this work could be used for a similar purpose if there is need for it.

As for the findings about the features affecting CPU loads, some findings from Kananen’s work were reinforced while others were contested. It should be noted that the features used in the two works are not identical, as some features were added or removed, and some were changed for this work. Features $x_2$ and $x_3$ were found to be the most influential in this work. In Kananen’s work these two features were represented by a single feature, which was also among the five most influential features.

Similarly, feature $x_7$ was found influential to CPU loads in both works. The differences come with features $x_1$, $x_5$, $x_6$ and $x_9$, which were not found significantly influential in this work, but were found significant in Kananen’s analysis. Especially feature $x_9$ was different, as in this work it was found almost completely meaningless for CPU loads. Additionally, $x_4$ was found significant in this work, while the equivalent feature was not significant in Kananen’s linear model. The differences can be partially explained by the different methods and metrics used for measuring importance and the fact that Kananen focused on C-RAN architecture whereas in this work it is just one of the architectures included in the samples for model 1. Still, features $x_2$, $x_3$ and $x_7$ can be said to have significant influence on SUT CPU loads after two separate works on the matter found them influential.

A secondary research objective in this work was to evaluate how well open-source Python data science libraries would perform in this kind of work. Overall, the libraries proved very useful. Pandas provided excellent labeled data structures that were logical, easy to manage and easy to modify. The various algorithms and data science methods that were at the core of this work came from the scikit-learn library. They were mostly easy to use with clear documentation, provided extensive optimization options and generally enabled good data science and machine learning analysis.

Most of the figures in this thesis are done with matplotlib library, which allowed for both easy and quick visualization and more polished and customized figure building. NumPy was used to a lesser extent but still made some mathematical operations easier. The fact that these libraries not only provide ready solutions for common data science tasks but also are fully compatible with each other and are built with each other in mind allowed the focus to be on actual analysis instead of re-implementing common solutions.
As was discussed during the analysis, one of the libraries did not perform up to par. Mlxtend was used to implement the sequential feature selection. The SFFS implementation uses algorithms and methods from scikit-learn to perform calculations, yet it provided conflicting and odd results that did not actually improve performance. While it is possible that this was a user error, there seemed to be no explanation for this in the API documentation.

Regardless of the implementation, using sequential feature selection could have been a mistake in hindsight. It is a greedy algorithm that is not guaranteed to find the best feature set. Using an exhaustive feature selection would have guaranteed finding the best feature set, although the number of iterations could have proved to be too big even for nine features. PCA could have been another option, as it would both reduce the number of features and remove the correlation between features.

Lastly, it is important to stress that majority of the work in the making of this thesis went into data exploration, data gathering and data wrangling. While this exploration was natural and necessary due to the explorative nature of the assignment, the fact that the system was not made with machine learning in mind meant that a lot of extra work was needed to normalize the data into a usable format. Making the system more compatible with data science would make machine learning efforts easier and faster and make the solutions more robust. The log parser used in this work to parse the data from the test logs will easily fail or parse wrong values if the test log printing is changed. Similarly, the log fetcher requires the run identifiers in the URL to fetch the correct logs, which is unique to each run of each job and is not easily obtainable.

7.1. Future work

As this work was meant to build a groundwork for automatic monitoring in the system, there is plenty of room for expansion and improvement. The most important aspect for future work in the environment is to create a better data plan to better support any future data mining and machine learning efforts. A consistent and well thought out data plan and structure would come a long way to help future automation efforts. Mapping what data is stored and where would save plenty of time for future data scientists. The databases that were mentioned at the beginning of this work are a good start for consistent and structured data storage and could be expanded further.

The created models could be improved further. As model 2 showed, eliminating feature $x_{10}$ by making it a constant improves the model performance. Splitting model 1 into multiple models as was done with model 2 could very likely improve overall performance of the monitoring tool, as it would improve homogeneity within the samples used to train each model. Additionally, the concept drift that was discussed during the analysis could be alleviated by removing older samples from the training set and retraining the models. Adding weights to training samples to emphasize newer samples over older ones could also be an option.

On the topic of model improvement, other regression methods could be explored to see if better models could be built with them. Multi-layered perceptron was tried in the explorative analysis and found inferior to random forest. However, this neural network was very unoptimized and run with relatively little data. Proper implementation of a
neural network with a framework built for deep learning like TensorFlow [67] could provide much better results. This would require investment into hardware more suitable for machine learning. Additionally, the features used in this work are not exhaustive of those used in test cases and adding or removing features could improve the models. The feature representation also was a compromise between simplicity and true representation of the feature space and could be revisited to better represent the more complex test cases.

The performance monitoring tool could be expanded by adding other predictive models into the decision module. During discussion of databases, anomaly detection was tried on one test job with good result, for example. CPU loads are not the only performance metric in the system and using the various other metrics for decision making could help make the tool more robust and catch a wider variety of problems. This would also allow extending the monitoring beyond the capacity jobs that were the focus of this work. Another way to expand the tools capabilities would be to make it real-time. Instead of outputting a report once a day, the tool could analyze a job’s results right after it has run and inform about possible problems immediately.
8. CONCLUSION

This thesis set out to explore possibilities of implementing machine learning in the capacity testing environment of a major telecommunications company that produces LTE standards compliant hardware and software. The aim was to see if machine learning could be used to automate monitoring and analysis of the test results of capacity testing. An additional goal was to explore the system and see how well it supported data science efforts. The effort was further motivated by the recent boom in machine learning and the availability of open source machine learning solutions. Open source Python machine learning and data science libraries were used in this work to free time for analysis instead of doing own implementations of algorithms and methods.

As the task and requirements were somewhat exploratory in nature, the majority of the work went into data mining efforts, such as locating, identifying and wrangling the data available. Multiple different data sets and machine learning methods were used, including anomaly detection, clustering and regression. In the end regression with a random forests algorithm provided the best potential with estimating CPU loads of hardware based on Layer 2 test parameters. Relevant parameters were determined with the help of previous work on the subject and expert knowledge.

Two separate predictive models were built to estimate the CPU loads. One model is a more general case model, while the other is more focused on a specific use case. Best hyperparameters for both models were determined through randomized and exhaustive searches through candidate hyperparameter sets. The models were trained with real test result data from the testing environment and validated with a separate set of real data. Both models performed well, with the more focused model performing better. Both mean squared error and absolute mean error rates were low, and the parameters could explain over 99% of the variation in CPU loads.

The two models are utilized in an automated performance monitoring tool. The tool is set up as a job in the testing environment, where it is run every day after major capacity tests are run to evaluate their results. The tool provides a report highlighting possibly anomalous test runs based on CPU loads. This report can help find problems in the tests and automates a part of the performance monitoring, freeing resources for other tasks.

Additionally, this thesis acts as a groundwork into machine learning in the testing environment. It highlights some possibilities for machine learning in the environment in addition to the regression that became the focus. It also shows the importance of a well-defined data plan for data science in a system like this. The test environment in this work needs better handling of data if further data science efforts are planned.
9. REFERENCES


10. APPENDICES

Appendix 1. Structure of the thesis process

Diagram:

1. Define problem
2. Explore system and available data
3. Choose and analyze data source
   - Suitable for the purpose?
     - Yes: Choose data representation and define scope
     - No: Continue exploring
4. Use a machine learning algorithm to create a model
5. Evaluate model
   - Model is valid?
     - Yes: Implement tool
     - No: Continue evaluating or return to data collection
Appendix 2. Feature importance’s of selected feature sets in feature selection for three different configurations with random forest.
Appendix 3. Plots showing development of absolute error rate for feature selection for random forest (left) and multi-layered perceptron (right)