MASTER'S THESIS 2022

Identification of relevant error descriptions in build logs using machine learning

Lykke Axlin, Klara Broman

ISSN 1650-2884
LU-CS-EX: 2022-52

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-52

Identification of relevant error descriptions in build logs using machine learning

Identifiering av relevanta felbeskrivningar i byggloggar med hjälp av maskininlärning

Lykke Axlin, Klara Broman

Identification of relevant error descriptions in build logs using machine learning

Lykke Axlin
ly6701ax-s@student.lu.se

Klara Broman
pol15kbr@student.lu.se

September 4, 2022

Master's thesis work carried out at Axis Communications AB.

Supervisors: Martin Höst, martin.host@cs.lth.se
Marcus Klang, marcus.klang@cs.lth.se
Gustaf Lundh, gustaf.lundh@axis.com
Ola Söder, ola.soder@axis.com

Examiner: Elin Anna Topp, elin_anna.topp@cs.lth.se

Abstract

Jenkins is the tool used at Axis Communications AB to facilitate the *Continuous Integration* (CI) pipeline. CI is the practice of frequently integrating code into a shared repository. To help the engineers find the lines containing relevant error descriptions in unstructured logs from failed Jenkins builds, a plugin called the Build Failure Analyzer (BFA) is used. The BFA utilises manually crafted regular expressions to scan the failed build logs, and rows that match a regular expression are highlighted. As new error descriptions arise in the build pipeline, new handcrafted regular expressions need to be added, which is a time-consuming process. Any error description that is not added to the BFA will not be caught and therefore the engineer will likely fail to notice it. Therefore, this Master's thesis aims to investigate to what extent it is possible to use historical data from the BFA to train an AI-model to perform text classification on build logs to detect relevant error descriptions that are not currently found by the BFA. Further, we intend to examine what requirements such a model needs to fulfill in order to be useful at the case company. To answer this question, an interview with the case company was conducted. In the interview, it emerged that the model is supposed to be used as a complement to the BFA. In the cases where the BFA does not find any error descriptions, the engineers can use the model's findings for a clue of where in the build log the relevant error description could be located.

To determine to what extent it is possible to train an AI-model to detect new error descriptions, a support vector machine (SVM) and an XGBoost model were trained on a dataset consisting of build logs annotated with historical data from the BFA, and evaluated on a manually annotated dataset. The SVM model achieved an average R-precision of 0.19, an F1-score of 0.60, and an MCC of 0.20. The XGBoost model achieved an average R-precision of 0.26, an F1-score of 0.57, and an MCC of 0.17. The results show that it is possible to detect new relevant error descriptions, although to achieve higher performance, further improvements to the dataset or a more advanced machine learning model is suggested.

Keywords: regular expressions, Jenkins, Build Failure Analyzer, build logs, text classification, support vector machines

Acknowledgements

We would like to thank our supervisors Martin Höst and Marcus Klang from Lund University for valuable feedback and guidance throughout the thesis. We also want to thank Gustaf Lundh and Ola Söder, our supervisors from Axis Communications AB, who have supported us and instructed us during this study. In addition, a thank you to Anton Friberg at Axis Communications AB, who also provided us with valuable input on our work.

Contents

Chapter 1
Introduction

When new code is committed at Axis Communications AB, the code is run through their build chain, which is a *Continuous Integration/Continuous Deployment* (CI/CD) pipeline, automated with open source CI/CD tool Jenkins. CI is the practice of frequently integrating code, while CD is the process of delivering and deploying code to production [24]. In the CI pipeline, code quality tests, unit tests et cetera are executed on a Jenkins server. By utilising CI and integrating small code changes frequently, bugs and errors are easier to find since new code is tested in smaller batches. When a build is run in Jenkins, log text with information about each module is produced. If a Jenkins build fails, an open source Jenkins plugin called the *Build Failure Analyzer* (BFA) [5] is triggered with the intention to find the rows in the build log most relevant to the errors that caused the build failure. The BFA uses manually crafted *regular expressions* and scans the build log to match any of them. A regular expression is a sequence of characters that is used to describe a pattern to search in a text [21, p. 35]. If any pattern matches a row, the row that contains the relevant error description is highlighted in the build log. However, if no regular expression matches are found in the build log, the engineers are left with no pointers on where in the build process the error occurred, and need to manually scroll through the build log. In order to keep the BFA updated, the engineers need to manually craft new regular expressions for each specific error, which is a time-consuming process.

Text classification is a task that belongs to the *Natural Language Processing* (NLP) field, i.e. the practice of analysing and categorising human language by combining knowledge from linguistics and computer science. To classify text implies to assign one or more classes to a text of any size, from a few words to an entire book. One classic example of text classification is spam detection, i.e. classifying e-mails as either *spam* or *not spam*. Text classification can be used on each line of a build log in order to find whether or not the line contains a relevant error description [21, p. 165].

This Master's thesis aims to investigate to what extent it is possible to use historical data from the BFA to train an AI-model to perform text classification on build logs to detect relevant error descriptions. Further, we intend to examine what requirements such a model

needs to fulfill in order to be useful at Axis Communications. We need to know how the solution will be used at the case company so that it can be integrated into their CI pipeline in the future.

1.1 Research questions

The purpose of this study is to provide answers to the following questions:

- To what extent is it possible to use historical data from the Build Failure Analyzer to train a machine learning model that can detect new relevant error descriptions in build logs?
- What requirements does the model need to fulfill in order to be useful at the case company?

1.2 Method

The study was performed according to the steps illustrated in Figure 1.1. To answer the first research question, data collection and machine learning experiments in three iterations were carried out. Each iteration consists of a data creation step, a preprocessing step, a training step, and finally an evaluation step. The iterations are explained in detail in Section 4.1. An interview with a Senior Engineer at the case company was conducted in order to answer the second research question.

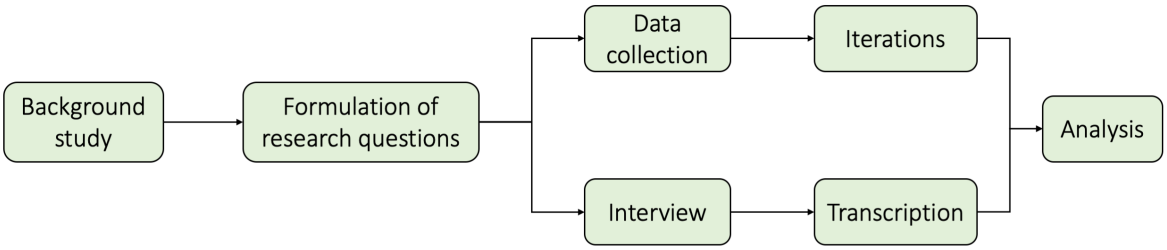

Figure 1.1: Overview of the research method

1.3 Contribution

In this Master's thesis we investigate and evaluate to what extent it is possible to use historical data from the BFA to train a machine learning model to detect new relevant error descriptions in build logs. The purpose of this is to evaluate if it is possible to use machine learning for this task. Further, we provide the case company with statistical information about the data, and give suggestions on how to better utilize it in the future.

Further, we investigate and give suggestions on how the model can be used in the build chain, in order for the developers at the case company to minimize the time spent on navigating the build logs.

1.4 Workload distribution

The implementation of the experiments were done by both authors. Most parts of the report have been written by both Lykke and Klara except for the following parts:

- Lykke: Method and results for the third iteration, created all the illustrations
- Klara: Method and results for the first and second iteration, Section 4.3 and 5.1 regarding the interview

Chapter 2
Background

This section covers the essential theory used in our project. It gives an overview of the CI/CD pipeline and how it is implemented at the case company. Furthermore, the current method of detecting errors in build logs at the company is described. Thereafter, the preprocessing of log files is described, followed by an overview of machine learning and the models used. Finally, the metrics used to evaluate our models are listed.

2.1 Case company

This study was carried out at Axis Communications AB. The company was founded in 1984 in Sweden and has over 3800 employees in 50 different countries. Initially, their main product was a print server, and in 1996 they launched the industry's first network camera. Today, they are a leading manufacturer of video surveillance systems [1].

This Master's thesis was completed at a department called R&D Tools at Axis. Since the R&D Tools department is responsible for the CI/CD pipeline, it is of great interest for them to improve the process for finding relevant error descriptions in the Jenkins build logs.

2.2 CI/CD pipeline

Continuous Integration (CI) is the procedure in which contributions from different software engineers are frequently integrated into a shared repository. This process is supported by a high level of automation: building, running code formatting checks, and more importantly; execution of unit and integration tests. By integrating as often as possible, preferably with small code changes, the risk of merge conflicts and breaking builds is minimised. Introduced bugs are easier to pinpoint and the problematic code can easier be lifted out of the code base, i.e. reverted. By leveraging the fact that the code is kept in a release-ready-state, organisations can ensure faster release rates [9].

In Figure 2.1, an overview of the steps in the CI/CD pipeline is illustrated. The first step after the code has been committed is called continuous integration. Here the code is built and run through both unit and integration tests. The following step is called *Continuous Delivery*, i.e. the practice of automatically preparing code changes for production. In the last step, called *Continuous Deployment*, the code changes are automatically deployed to production [24].

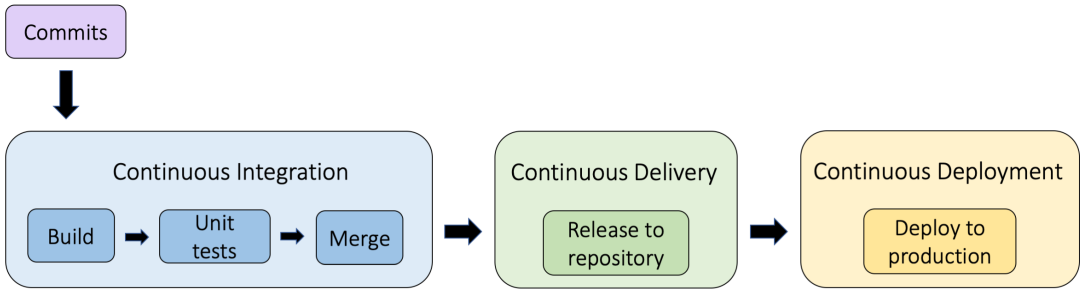

Figure 2.1: Overview of the steps in the CI/CD pipeline

2.3 Jenkins and Build Failure Analyzer

Jenkins is an open source automation server that can be used to facilitate CI/CD pipelines [12]. There are hundreds of plugins that can be used to configure Jenkins. An example of a plugin is the *Build Failure Analyzer* (BFA), which is the open source tool currently used at the case company to find relevant error descriptions in build logs [5].

If a build fails, the BFA is triggered and starts scanning the build log with the manually crafted regular expressions it has configured. If matching relevant error descriptions are found, the corresponding lines are highlighted, as seen in Figure 2.2. This makes the debugging easier for the engineer. If the file only contains error descriptions that are not configured in the BFA, no lines are highlighted, so the engineer will have to manually go through the unstructured logs of hundreds and sometimes even thousands of lines in search for the error description. In order to keep the BFA efficient, regular expressions for new errors need to be added continuously as new errors occur. Engineers need to set up the BFA by manually adding each relevant error description they want the BFA to detect, and its corresponding regular expression pattern.

An example of the metadata that is saved when the BFA finds relevant error descriptions in a build log is presented in Figure 2.3. The `indications`-list contains all the `pattern`-`matchingString` pairs found by the BFA, where the `pattern` value represents the regular expression, and the `matchingString` value is its matching line in the build log. Only the first match of each pattern is saved as a `matchingString` value in the `indications`-list, even though a pattern might match multiple lines in a build log.

```
=== Running task for junit ===
[Pipeline] junit
Recording test results

No test report files were found. Configuration error?
[Pipeline] }
[Pipeline] // stage
```

Figure 2.2: Highlighted relevant error description by the BFA in a Jenkins build log


```
{
  "failurecauses": [
    {
      "name": "No test report files were found",
      "description": "This is usually a configuration error […]",
      "id": 0,
      "indications": [
        {
          "pattern": ".*No test report files were found. Configuration error?.*",
          "matchingString": "No test report files were found. Configuration error?"
        }
      ]
    }
  ]
}
```


Figure 2.3: The JSON-object containing information about relevant error descriptions found by the BFA

2.4 Machine learning

Machine learning is the process of using data and algorithms to teach computers the ability to learn without being explicitly programmed to do so [13, Chapter 1]. In this section, the machine learning models used in our study will be described.

2.4.1 Supervised and unsupervised learning

Supervised and *unsupervised learning* are two different ways to train machine learning models. The main difference is that in supervised learning the model is trained on labeled dataset, while in unsupervised learning the training data is unlabeled. The goal with supervised learning is to predict outcomes for new data. Supervised learning is often used for text classification problems, such as identification of unwanted spam messages in e-mail or classification of errors in log files. A support vector machine is an example of a model commonly used for supervised learning problems [14, p. 19-21].

In unsupervised learning, the model decides what information is relevant. Unsupervised learning is commonly used for grouping data, for example to detect anomalous behaviour through identifying patterns that deviate from the known groups [14, p. 286].

2.4.2 Logistic regression

Logistic regression is a supervised machine learning algorithm that can be used for binary classification problems. The algorithm uses a logistic function:

$$f(x) = \frac{1}{1 + e^{-x}}, \text{where } f(x) \in [0, 1] \tag{2.1}$$

to calculate the probability of the input x belonging to either class 0 or 1.

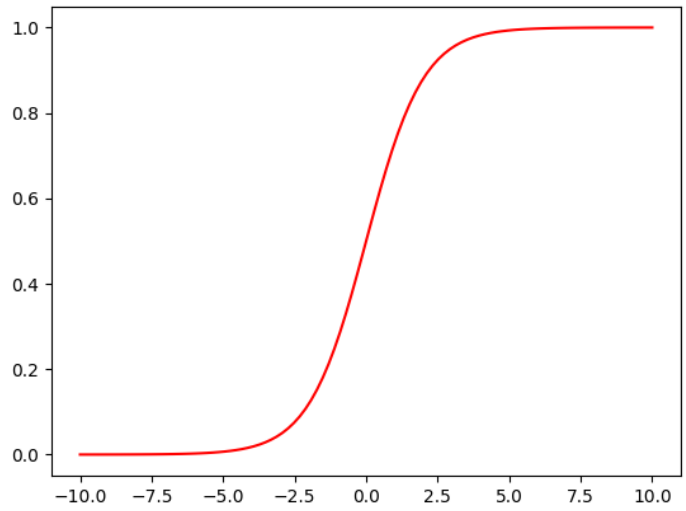

Figure 2.4: The Logistic regression function

The logistic function has an S-shaped curve as seen in Figure 2.4, and for outputs from the logistic function greater than 0.5 a class 0 is predicted, otherwise a class 1 is predicted [3, p. 185].

2.4.3 Support vector machines (SVM)

The main principle of *Support vector machines* (SVM) is to separate data points in a multidimensional space by creating a *hyperplane* between them, i.e. a flat boundary [14, p. 239-242]. In the simplest case of two dimensions, the hyperplane corresponds to a line that separates classes of linearly separable data points. Starting from data annotated with class labels, the SVM decides the position of the line by finding the *maximum margin* hyperplane between points, i.e. the line that is midway between the two classes. This is equivalent to maximising the distance between *support vectors*. Support vectors are the data points of each class that are closest to the line, as shown in Figure 2.5. Calculating the maximum margin assures that the datapoints will stay on the correct side of the hyperplane despite random noise [14, p. 241].

The SVM algorithm can be adjusted to perform both classification and prediction of numerical values and text classification is one of the applications where it has proved to be successful [14, p. 239-247].

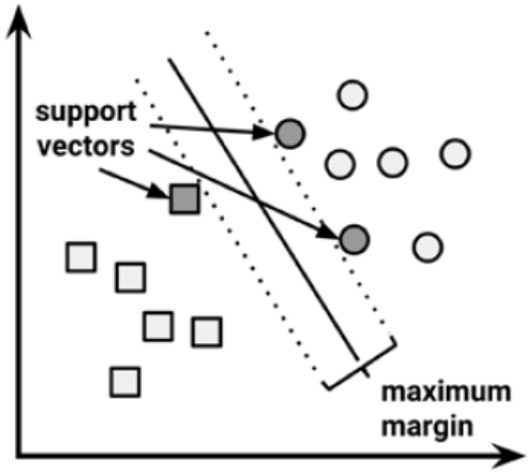

Figure 2.5: Classification of data points with SVM in two dimensions [14, p. 242]

2.4.4 Gradient boosting

Gradient Boosting is an *ensemble learning* model that combines multiple *decision trees*. A decision tree is a tree structure algorithm where each node represents a decision and the leaf nodes are the final class labels [14, p. 126], and an ensemble learning algorithm combines multiple models to make a prediction [14, p. 359]. In our study the XGBoost [7] implementation of Gradient Boosting is used. XGBoost resamples the data to generate complementary decision trees. The prediction of each tree is weighted based on its past performance so that better performing models have more power over the final prediction [14, p. 366].

Some advantages of the XGBoost implementation are that it is fast compared to other similar models, and scalable so that data with hundred millions of examples can be processed on a desktop [7].

2.4.5 Neural networks and deep learning

An artificial neural network is a computational model that mimics the behaviour of neural networks in the brain. It can be thought of as a directed graph where the nodes represent neurons and the edges represent synapses between them [26, p. 228]. The nodes of the graph are grouped in connected layers. In each layer the input from the previous layer is multiplied with weights and sent to the next layer. Layers between the input and output layer of a network are called hidden layers, and networks with multiple hidden layers are called *deep neural networks*. Deep neural networks are better at modelling more complex problems than traditional machine learning algorithms [14, p. 226-230].

2.4.6 Hyperparameter tuning

Hyperparameters are parameters that are not learnt by the estimator during training. They are instead static and passed in as arguments to the estimator and are used to control the

learning process. In contrast to hyperparameters, there are parameters that are learnt through training, for example node weights where the value is derived during training [15].

For some machine learning problems the base models perform well, but for others, it is necessary to find optimal values to hyperparameters to get good performance. Since it is a time-consuming task to manually try out different sets of hyperparameters, it is usually recommended to utilise search techniques to find the optimal set of parameters [14, p. 348]. However, hyperparameter tuning can be very computationally costly, and it is not guaranteed that high performance will be achieved. The tuning process involves trying out all, or many, different sets of parameters. Additionally, large and complex datasets can further affect the tuning cost negatively [20].

2.5 Preprocessing

All the models used in this study require input in the form of numerical vectors. For example, the SVM classifies data through separating the data points with a hyperplane, as described in Section 2.4.3. Therefore, when using natural language processing the text in the corpora needs to be transformed to numerical vectors before it can be used as input to the model [21, p. 65]. In this section, we discuss the methods we used to transform and improve the data before using it to train a model.

2.5.1 Term frequency-inverse document frequency (TF-IDF)

An approach that is commonly used to vectorise the text is called *term frequency-inverse document frequency* (TF-IDF). The idea is to give each word a weight based on how relevant it is to a certain corpus relative to all the other documents in the corpora. In other words, words that occur often in all of the documents are given low weights since they do not contribute to the actual content of the document, this could for example be stopwords such as *it*, *our* and *have*. On the other hand, if a certain word has a high frequency in a particular document, the word is given a high weight since it is unique and relevant to that document [13, Chapter 6].

The TF-IDF of a document is calculated by multiplying the *term frequency* (tf) by the *inverse document frequency* (idf). The tf is the number of occurrences of the word in a document and the idf is a measurement of how unique the word is in the whole corpus. The formula is given by:

$$\text{tfidf}(t,d,D) = \text{tf}(t,d) \cdot \text{idf}(t,D) \tag{2.2}$$

where t is the word, D is the corpus and d is a document in the corpus [28].

In Table 2.1, an example of a TF-IDF *feature matrix* generated from the build log lines `ERROR: Could not check out resource` and `ERROR: Failed trigger` is presented. The feature matrix consists of the features and their corresponding weights in the different sentences. As seen in the table, the term `ERROR` occurs in both sentences but gets a different weight due to the sentences having different number of words.

Table 2.1: TF-IDF vector representation of two sentences from Jenkins build logs

	check	could	error	failed	not	out	resource	trigger
1	0.43	0.43	0.30	0.00	0.43	0.43	0.43	0.00
2	0.00	0.00	0.45	0.63	0.00	0.00	0.00	0.63

2.5.2 Dimensionality reduction

Dimensionality reduction is a technique to downscale the dimension of the feature matrix produced in the vectorisation step, in this study the TF-IDF vectoriser is used. This can be useful since a large number of features can prolong the training process of a machine learning model. Additionally, the data might contain non-relevant features that can cause a model to overfit, i.e. miss the important pattern in the data and therefore not generalise well to new cases [14, p.15]. When dimensionality reduction is added, only the most important features of the feature matrix are kept [26, p. 278].

2.5.3 *N*-grams

When looking at the frequency of tokens in the corpus it can be useful to include the neighbouring features to study the context in which the token appears in. Unigrams look at the frequency of one single word at a time, bigrams look at the frequency of two adjacent words and trigrams look at the frequency of triples and so on. A more general denotation is called an *n*-gram where *n* number of words in a sequence are analysed [21, p. 135]. An example of a sentence with unigrams, bigrams, and trigrams is presented in Table 2.2.

Table 2.2: Unigram, bigram, and trigram example

n	Features
1	No, test, report, files, were, found
2	No test, test report, report files, files were, were found
3	No test report, test report files, report files were, files were found

2.5.4 Data resampling

A dataset is considered imbalanced if there is an uneven distribution between the classes in the dataset. The problem with imbalanced data is that it can lead the model to only predict the majority class as it tries to maximise classification accuracy [23]. For example, let the minority class make up for one per cent of the whole dataset and let it represent rows containing relevant error descriptions. Then the model can always predict that the row does not contain any relevant error description, and achieve a high accuracy even though the model is not useful for detecting relevant error descriptions [14, p. 312].

One way to handle imbalanced datasets is to undersample the majority classes. The most naive approach is to use a technique called random undersampling which randomly deletes

entries of the majority classes until the desired ratio between the classes is achieved. The resampling is only applied on the training set and can be very effective depending on the data, although there is a risk that important information is lost in the process [16].

2.6 Metrics

In this study, several different metrics were used to compare the performance of the models. In this section, they are listed and described.

2.6.1 Basic metrics

In binary classification, *true positive* (TP) and *true negative* (TN) are used to denote when a condition or characteristic has been correctly identified as positive or negative respectively. In anomaly detection a TP could for example represent that the model correctly identified an anomaly and a TN would then represent that the model correctly determined that the document did not contain an anomaly, i.e. the document was ”normal”. In the same example, a *false positive* (FP) represents that the model incorrectly classified the document as anomalous, and a *false negative* (FN) that the model incorrectly classified an anomalous document as normal [13, Chapter 1]. Combining these four variables into a matrix is called a *confusion matrix*, which is illustrated in Figure 2.6 [21, p. 213].

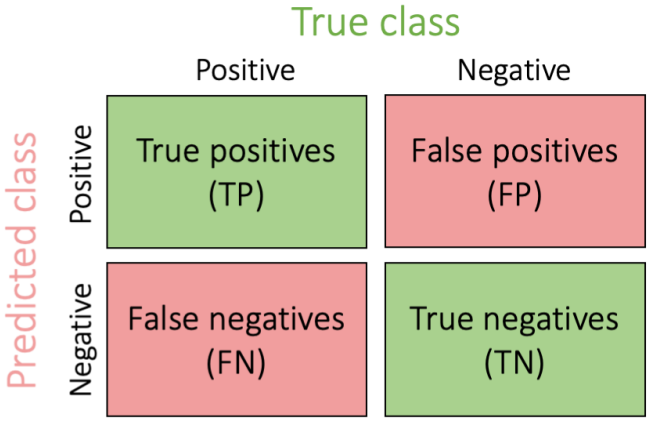

Figure 2.6: Confusion matrix for two classes

2.6.2 Accuracy

Accuracy is a statistical metric used to determine the proportion of predictions the model correctly classified:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \text{where accuracy} \in [0, 1] \tag{2.3}$$

2.6.3 Precision

Precision measures the proportion of the detected items that were relevant:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \text{where precision} \in [0, 1] \tag{2.4}$$

2.6.4 Recall

Recall measures the proportion of relevant items that were detected:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \text{where recall} \in [0, 1] \tag{2.5}$$

2.6.5 F1-score

The F1-score is the harmonic mean of precision and recall of a model and is often used to compare different models to each other and is calculated as:

$$\text{F1-score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}, \text{where F1-score} \in [0, 1] \tag{2.6}$$

2.6.6 Matthews correlation coefficient (MCC)

Another way to compare performance of different models is to use the *Matthews Correlation Coefficient* (MCC) calculated by the following formula:

$$\text{MCC} = \frac{\text{TP} \cdot \text{TN} - \text{FP} \cdot \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}}, \text{where MCC} \in [-1, 1] \tag{2.7}$$

The closer to 1 the MCC score is, the more correct are the predictions. A value around 0 is as good as random predictions, and values close to -1 mean almost complete difference between predictions and observations. The main difference between F1-score and MCC is that MCC summarises all parts of the confusion matrix while F1-score leaves out the true negatives. This makes MCC more informative than F1-score and more suitable when used for evaluating binary classification on imbalanced data [8].

2.6.7 Average R-precision

The average R-precision is calculated by the following formula:

$$\text{ARP} = \frac{1}{\text{n}} \sum_{\text{n}} \text{RP}_{\text{n}} \tag{2.8}$$

where n is the number of topics, and RP is the R-precision value for a particular topic [2, p. 195]. In our case, a topic corresponds to a log file. To calculate RP, the rows of each log file are ranked by the probability that the model will predict the row as a relevant error description. An example of this is given in Table 2.3 below.

Table 2.3: Example of calculation of average R-precision in a log file

n	Row number	Probability	Relevant
1	46	0.89	x
2	198	0.76	x
3	43	0.75	
4	590	0.50	x
5	4	0.43	
6	1000	0.23	x
7	456	0.10	x
8	90	0.08	x

In the example, eight rows from a log file are ranked. R represents the number of relevant error descriptions, in this case $\boldsymbol{R = 6}$, since the file contains six relevant error descriptions. Four of them were ranked among the six most probable to be found by the model. This results in an R-precision of $\frac{4}{6} \approx \mathbf{0.67}$ for this log file. This metric differs from normal precision, described in Section 2.6.3, since the R-precision only represents the precision of predictions with probabilities above a certain threshold.

Chapter 3
Related work

Automatic analysis of logs using machine learning is a well-known research problem that has been addressed in various insightful studies that propose different solutions to this problem. In this section, we give a brief summary of the studies that have inspired our work. The logs analysed in the different studies are both logs from different continuous integration tools, and other types of logs, such as system logs.

3.1 Build log annotation

Brandt et al. [4] provide a collection of 800 manually annotated logs from Travis CI, which is another tool used for CI, from 80 GitHub repositories including 29 main development languages. In the dataset described in the paper, each build is represented by a chunk, i.e. the lines from the build log that contain the information about why the build failed. Additionally, each chunk is assigned a structural category that represents how the chunk is formatted in the build log.

3.2 Model selection

Studiawan et al. [27] proposed a sentiment analysis technique to detect anomalies in log files. Sentiment analysis is a form of text classification used to measure the attitude expressed in a text, for example it can be used to determine if a customer review is positive, negative or neutral. According to the study, deep machine learning models have shown improved results compared to classical machine learning algorithms when used for log anomaly detection. A comparison between different models was carried out, where the authors' model achieved the best performance with an F1-score of 0.99 and an accuracy of 0.99. Among the other evaluated models, the supervised ones generally showed higher performances compared to the unsupervised ones.

3.3 Anomaly detection and error classification

In the Master's thesis by Mandagondi [19] anomaly detection in log files from *Java Common Auto Tester* (JCAT) using machine learning was studied. Three ML algorithms were evaluated: *Local outlier factor*, *Random forest* and *K-means clustering*. Local outlier factor is an algorithm that finds anomalous data points by measuring how similar a data point is to its neighbours. K-means clustering combined with data preprocessed with TF-IDF resulted in the best performing model with an F1-score of 0.91. K-means clustering is an unsupervised ML technique that can recognise groups of similar data points in a dataset. The Random forest algorithm combines several Decision trees and outputs the class that was selected by most trees.

In another Master's thesis, Lindqvist [18] used machine learning to classify build failures in Jenkins logs as either *infrastructure error* or *user error*. The study concluded that the human factor is still needed to create regular expressions that can classify the error descriptions, but that machine learning can be used as a powerful tool to make predictions of unknown failures. However, one could try to only use failed build steps to prevent misleading the models with successful build steps. The author also found that some of the training data contained errors that had been wrongly classified by the regular expressions. TF-IDF was used for preprocessing and three ML algorithms were evaluated in the study: support vector machine, random forest and gradient boosting classifier. The best performing one was the gradient boosting classifier with an F1-score of 0.79.

We combined the methods of these two Master's theses to locate the rows containing relevant error descriptions in the log file. Like Mandagondi [19] we implemented anomaly detection in log files but we were also interested what part of the log that contained an error and not just if there was an anomaly. While Lindqvist [18] classifies the category an error belongs to, we used text classification to determine whether a build log line contains a relevant error description or not.

3.4 Log file classification pipeline

Catovic et al. [6] described their implementation of log file classification used in large-scale industry projects called *Linnaeus*. The implementation consists of a highly configurable pipeline that can be used for both anomaly detection and multilabel classification of eventual errors in various types of logs. Linnaeus automatically chooses a suitable classification model from a number of classical ML algorithms such as: Gaussian naive Bayes, SVM, Logistic regression and Random forest. This paper served as a practical example on how a model could be integrated into a continuous integration pipeline in an industry setting.

Chapter 4
Method

In this section, the approach to answering the research questions is described. Further, the interview and the methodology of each step of the machine learning pipeline is described, as well as the tools needed to perform the respective tasks.

4.1 Overall approach

This study was conducted according to the research method presented in Figure 1.1. Initially, a background study of related work was performed and the research questions were formulated.

The method used to answer the first research question was an action research method, which is a research methodology where problem solving is performed through thoroughly documented activities [11]. In our study, the action research consisted of a data collection step along with three iterations of a machine learning pipeline, as illustrated in Figure 4.1. The purpose of an iterative approach was to start off simple and then methodically make changes to the pipeline as we gain knowledge, and evaluate how the changes impact the performance. Each iteration of experiments was conducted with a machine learning pipeline consisting of a data creation step, a preprocessing step, a training step, and an evaluation step. Between the iterations, changes to the dataset creation step or the evaluation step or both were introduced.

To answer the second question, an interview with the case company was conducted. The interview process is described in Section 4.2.

4.2 Interview

In order to find out more about how the BFA is used at the case company and what requirements they have on an AI-based system that can detect relevant error descriptions, an interview was conducted. An interviewee was selected among the group of engineers working

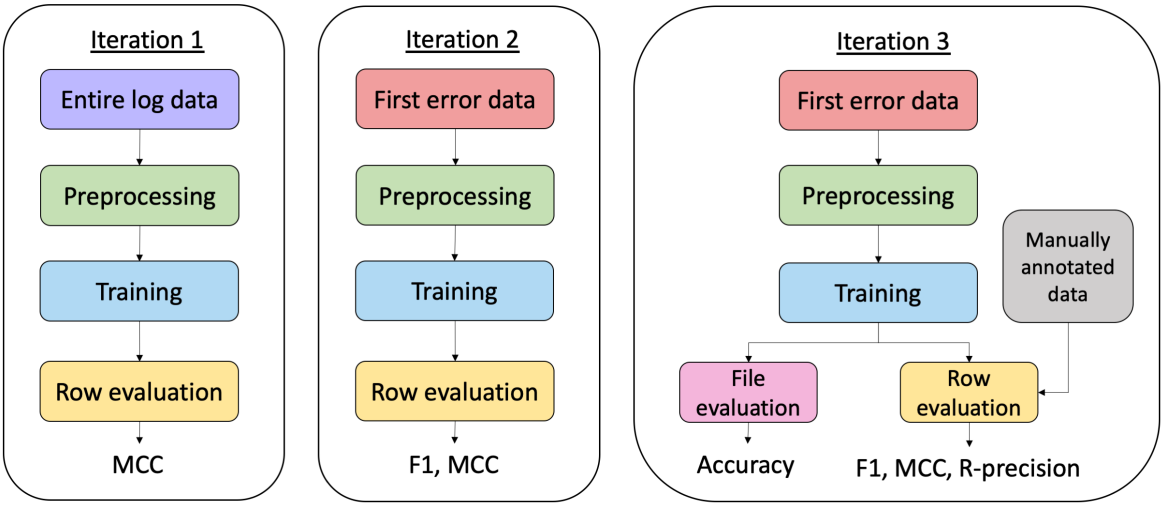

Figure 4.1: Overview of modifications between the three iterations

with the maintenance of the Jenkins servers. The engineer selected is titled Senior Engineer specialised in developer experience and has many years of experience in CI. A semi-structured interview was carried out, where two interviewers took turns in asking the questions listed in Appendix A. The interview was recorded, transcribed and summarised in Section 5.1.

4.3 Data collection

In order to create the datasets used in the iterations, Jenkins build logs as well as metadata from the BFA had to be collected. In this section, the build logs and how they were collected are described.

4.3.1 Jenkins build logs

The logs generated during Jenkins builds contain information about the build, such as compilation results and reports from unit and integration tests. At the case company, there are around ten servers running Jenkins builds. Build logs from five of the servers, named server 1-5 here, were included in this study.

The appearance and structure of the logs vary depending on what type of build is executed. On server number 1 and 5, users can set up their own builds. Builds of the case company's own operating system run there, as well as other builds of software written in both Java, C, and Python. Server number 2 is a large build server that mostly runs builds of the case company's own operating system. The builds that run on server number 3 run both unit, integration, and build tests. All builds are set up with the same configuration, and the server is owned by the Tools department. Server number 4 is used for building and signing, and just like on server number 3, the configuration is coherent between builds.

4.3.2 Collecting data

The build logs are stored on the Jenkins servers, and metadata about the builds, such as timestamp, status, and url, is stored on an Elastic server. Elastic is a search and analytics engine used at the case company [10]. In the *Save logs* step in Figure 4.2, the Elastic server is queried for metadata about all failed Jenkins builds within a specified date range. For each failed build, the log text is downloaded from the Jenkins server the build was run at. Two files are saved, one containing the console text and one containing metadata from Elastic saved in a JSON-object. For logs where the BFA has found errors, the metadata file contains a list of matching error entries, with the corresponding regular expression and matching string of each error found in the log text.

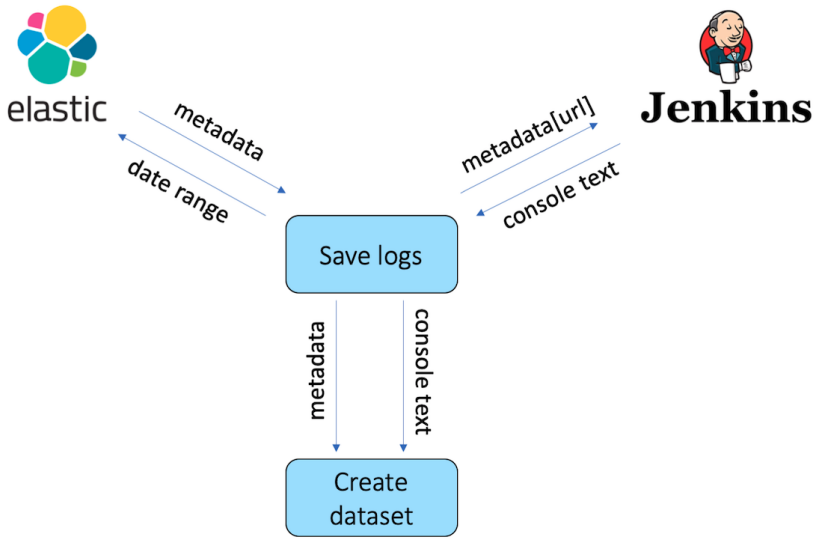

Figure 4.2: Illustration of the steps to collect data and create the dataset

4.4 Dataset creation

The creation of the dataset was carried out by annotating each log line of the failed Jenkins build logs using metadata from the BFA. Two different approaches of annotating the data were used and evaluated. In this section the Jenkins build logs, the data collection process, and the two annotation techniques are described.

4.4.1 Dataset structure

In the *Create dataset* step in Figure 4.2, each build log line was combined with the corresponding metadata object from Elastic to create the dataset. During this study, two different datasets were created and used to train and evaluate the models. In the last iteration, a manually annotated dataset, described in Section 4.4.1, was created to evaluate whether the models could predict new relevant error descriptions. All datasets follow the same structure with the

columns shown in Figure 4.3 and 4.4. Each `row` number refers to the row number in the build log. The `is_faulty` column indicates whether the corresponding `line` contains a relevant error description or not. Lines annotated with 1 contain relevant error descriptions, while lines annotated with 0 do not. For lines containing relevant error descriptions, the matching regular expression patterns are saved in the `regex` column. The `server` column indicates which server the corresponding build was run on, and the `file_id` represents the file name of the file where the build log text was saved.

The dataset described by Brandt et al. [4] consisted of manually annotated chunks from build logs, as described in Section 3.1. In our study, we created two automatically annotated datasets and one manually annotated evaluation dataset that was used in the third iteration. The reason that we did not annotate the training logs manually in our study is that we wanted to investigate whether historical data from the BFA would be sufficient to train a model to find relevant error descriptions. Since the BFA saves matching lines, it was more convenient to split the log files into single lines, although it is possible that chunks would have provided more context and therefore resulted in a better model.

Entire log annotation method

In the first iteration, an annotation method we call the *entire log annotation method* was used. The name refers to that all rows in each build log are included in the dataset. The method makes use of the metadata, shown in Figure 2.3, which contains a list of `indications`, i.e. error descriptions that the BFA has found in the corresponding log file. Each indication contains the regular expression pattern and the `matchingString` field representing the line in the log file where the match of the pattern was found. With this annotation method, all indications from the BFA were used to annotate the log lines. In the *Create dataset* step in Figure 4.2, each log line was compared to the `matchingString` fields of the metadata. If the log line corresponded to one of the matching strings, it was annotated with 1 in the `is_faulty` column, otherwise it was annotated with 0. In Figure 4.3, an example from the dataset created with this annotation method is presented.

Only the first matching line for each regular expression is saved in the BFA's indications-list, and thus only the first matching line for each regular expression was annotated with a 1. However, it is possible that there are more lines in the log files that match the regular expressions, but that are not saved in the `indications` list. With this annotation method, those lines are annotated with a 0, even though they contain relevant error descriptions. To avoid including relevant error descriptions annotated with 0, another annotation method was tried in the second and third iteration.

row	is_faulty	file_id	server	regex	line
49	0	10	1	None	=== Running task for junit ===
50	0	10	1	None	[Pipeline] junit
51	0	10	1	None	Recording test results
52	1	10	1	.*No test report files were found. Configuration error?.*	No test report files were found. Configuration error?
53	0	10	1	None	[Pipeline] }
54	0	10	1	None	[Pipeline] // stage
55	0	10	1	None	[Pipeline] publishMQMessage

Figure 4.3: Example rows from dataset created with the entire log annotation method

First error annotation method

In the second and third iteration, an annotation method we call the *first error annotation method* was used. When creating the dataset with the first error annotation method, only the first relevant error description found by the BFA in each file was included. Each line was compared against the `pattern` fields from the BFA until a match was found. All lines following the first match were omitted. An example from the dataset annotated with the first error annotation method is presented in Figure 4.4.

This method resulted in a smaller dataset than the entire log annotation method, but with less risk of including wrongly annotated lines in the dataset.

row	is_faulty	file_id	server	regex	line
49	0	10	1	None	=== Running task for junit ===
50	0	10	1	None	[Pipeline] junit
51	0	10	1	None	Recording test results
52	1	10	1	.*No test report files were found. Configuration error?.*	No test report files were found. Configuration error?
0	0	11	2	None	Started by timer
1	0	11	2	None	Running as SYSTEM
2	0	11	2	None	Posting JSON message to RabbitMQ:

Figure 4.4: Example rows from dataset created with the first error annotation method

Manual annotation method

A manually annotated dataset was needed in the final evaluation step in the third iteration of experiments. The reason for this was that it is not possible to determine whether a model can detect new relevant error descriptions that have not been found by the BFA, with a dataset annotated with historical data from the BFA. The dataset was created from 100 failed build logs where the BFA had not found any relevant error descriptions. The dataset was manually annotated by two senior engineers at the case company with domain knowledge of Jenkins. The entire logs were included in the dataset and all lines with relevant error descriptions were annotated with 1.

4.4.2 Dataset statistics

In total, 48 724 build logs from 61 days were collected in the data collection step, as described in Section 4.3.2. Among those build logs, there were 35 311 logs from failed builds where the BFA had found at least one relevant error description. In other words, the BFA was helpful in approximately 73% of the cases. The smaller datasets produced from 15 days of builds, consisted of data from 1 757 logs. In total, the size of all logs collected from 61 days is 42GB, and the size of the build logs collected from the 15 first days is 2GB. The average log file size of the build logs collected is 937kB.

In Table 4.1, the distribution of files containing at least one error found by the BFA between the different Jenkins servers can be found. As seen in the table, the majority of the errors come from either the first or second server.
The distribution of 0:s and 1:s in the respective datasets can be observed in Table 4.2. Both a large dataset with data from all 61 days, and a smaller with data from 15 days were created

Table 4.1: The number of build logs from each server

Server	1	2	3	4	5
Number of logs	16 810	16 344	1 247	875	35

with the entire log annotation method. Only 15 days of data was used for the dataset created with the first error annotation method, because as seen in Section 5.2.2, it was not feasible to train the model on the dataset from 61 days.

Table 4.2: Distribution of 0:s and 1:s in the differently annotated datasets

	Entire log annotation		First error annotation	
Days	**1**	**0**	**1**	**0**
15	7 723	18 880 731	1 734	17 617 095
61	187 770	460 197 174	-	-

In Figure 4.5, the number of matching relevant error descriptions for the different regular expressions in the dataset annotated with the entire log method is visualised. As seen in the figure, there are a few regular expressions that make up for almost all the relevant error descriptions found by the BFA. For example, we can see that only two regular expressions have more than 1000 matches in the data, while 24 regular expressions only have less than 100 matches in the corpus. Likewise, in Figure 4.6, the regular expression distribution for the dataset annotated with the first error method is illustrated. As seen in the figure, there are fewer matches of regular expressions in this dataset, but the distribution between the regular expressions is more or less kept.
The manually annotated dataset that was created for the final evaluation consisted of logs from 100 builds where the BFA had not found any relevant error descriptions. The distribution between 1:s and 0:s in that dataset is presented in Table 4.3.

Table 4.3: Distribution of 0:s and 1:s in the manually annotated dataset

	0	1	Total
Number of rows	60 447	646	61 093

4.5 Preprocessing

In this section, the data split methods as well as the tools used in the preprocessing step are described.

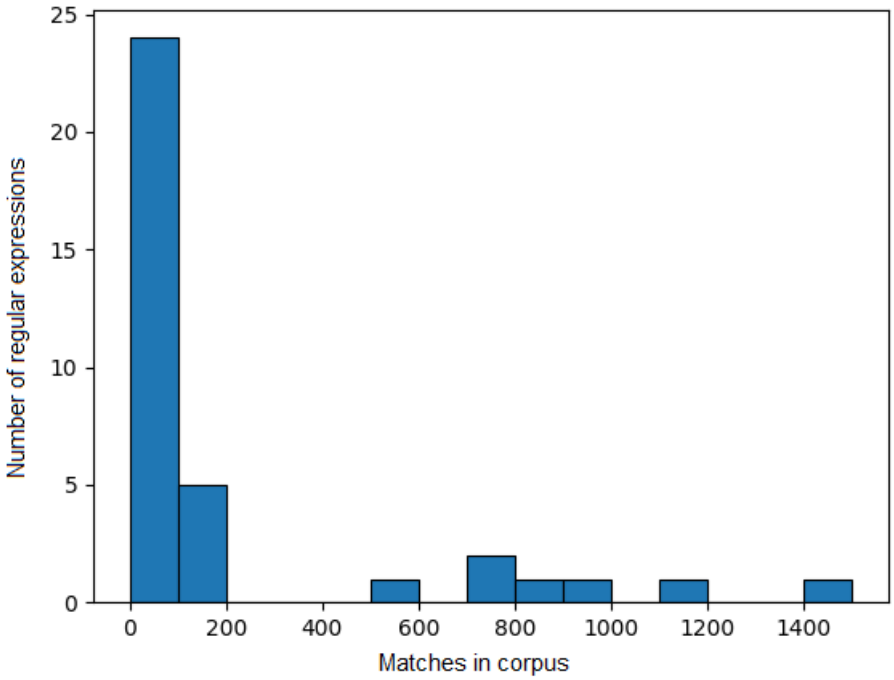

Figure 4.5: Distribution of the different regular expressions in the 15 days dataset annotated with the entire log method

4.5.1 Data split and stratification

Before training and evaluating the models, we split the data into one training and one test set. This ensures that the model is tested on data it has not been trained on. In our experiments, three different data splitting techniques were used: random split, stratification based on server, and stratification based on regular expression.

To determine whether the model could detect error descriptions it had already been trained on, experiments with randomly split datasets were carried out in all three iterations. The distribution between the training and test set was 80/20.

In the first iteration of experiments, we tried using a stratified split based on data source, i.e. the Jenkins server that the log file was downloaded from. The initial thought behind this split was that the data from different Jenkins servers would look different. The test set consisted only of log files downloaded from server 1, and the training set consisted of log files from the rest of the servers. However, many of the regular expressions were still present in both the training and test set. Additionally, since 16 810 build logs were collected from server 1, and 18 501 build logs were collected from the rest of the servers, as shown in Table 4.1, the distribution of logs between the train and test sets was almost 50/50.

Two different approaches were used in order to evaluate if the model could find errors it had not previously been trained on. In the first iteration, a stratified split based on regular expressions was carried out. To achieve this, a random 20% of the regular expressions were picked. When creating the dataset, all rows annotated with 1:s with regular expressions that matched any of those 20%, were put in the test set and the remaining were put in the training

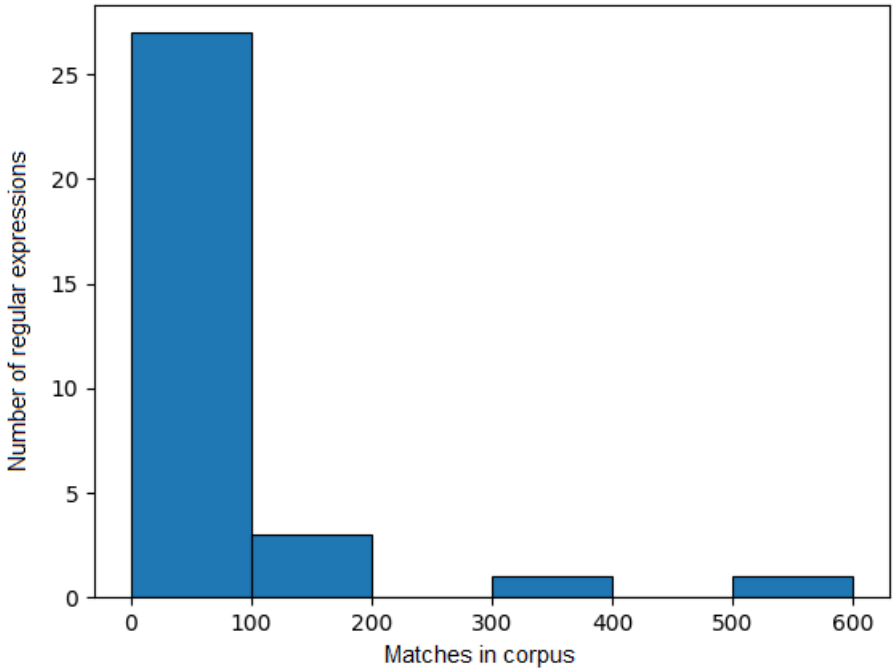

Figure 4.6: Distribution of the different regular expressions in the 15 days dataset annotated with the first error method

set. The rows annotated with 0:s were also split across the training and test sets with a 80/20 distribution. In the third iteration, a manually annotated dataset was used for evaluation.

4.5.2 Text vectorisation

The `TfidfVectorizer` from the scikit-learn library [22] was used in all three iterations to vectorise the text, as described in Section 2.5.1. In the two first iterations, it was used with default parameters, and in the last iteration, different values of the following parameters were tested:

- `min_df` - A value of 0.1 filters out all the terms that do not exist in at least 10% of the documents in the corpus.
- `max_df` - A value of 0.9 filters out all the terms that exist in more than 90% of the documents in the corpus.
- `ngram_range` - Controls what n-grams to use. For example, with the input value (1,2) both unigrams and bigrams are used.

4.5.3 Over- and undersampling techniques

Since the distribution of regular expressions in the data was uneven, as seen in Figure 4.5 and 4.6, a `RandomOverSampler` from the imbalanced-learn library [16] was used to copy the less

frequently occurring regular expressions in order to achieve a more even distribution. The mean frequency of the regular expressions was calculated and all regular expressions with less occurences than the mean were resampled to the mean. Oversampling was tested in both the first and the third iteration.

To avoid the problems that can arise with an imbalanced dataset, as described in Section 2.5.4, undersampling of the rows annotated with 0:s was performed using a `RandomUnderSampler`. The RandomUnderSampler randomly removes rows until the distribution of classes corresponds to the `sampling_strategy`. Several different sampling strategy values were tested in all three iterations. A sampling strategy of 0.5 means a ratio of 1:2 between 1:s and 0:s, and 1 means a ratio of 1:1.

4.5.4 Dimensionality reduction

In order to perform dimensionality reduction, i.e. reduce the dimensions from the matrix generated by the TF-IDF vectoriser, as described in Section 2.5.2, we used a `TruncatedSVD` from scikit-learn [22] in the third iteration of experiments. The TruncatedSVD has a parameter called `n_features` which determines the dimension of the output matrix.

4.6 Models

Supervised machine learning models performed better than unsupervised models among the classical models evaluated in the study by Studiawan et al. [27] One of the supervised models evaluated in the study was the SVM, which is commonly used for text classification problems [14, p. 240]. Therefore, we chose to start off with an SVM model.

Initially, our intention was to perform experiments on a deep learning model and compare with the results from the SVM. Due to time constraints, we did not experiment with any deep learning models. Instead, an XGBoost model was trained and compared with the SVM in the third iteration since in the study by Lindqvist [18], the gradient boosting model achieved better results than the SVM.

The SVM implementation, called `SVC`, from the scikit-learn library [22] was used in all three iterations. In the first iteration, hyperparameter tuning of the following parameters was carried out:

- `C` - Regulates the trade-off between classifying all training examples in a dataset correctly and more generalisation, i.e. a smoother decision boundary between the classes. A lower value results in more examples classified correctly, but with an uneven decision boundary and smaller margin. With a higher value, the decision boundary will be smooth with a larger margin, but more examples will be wrongly classified [14, p. 245].
- `gamma` - Controls how much each single training example impacts the model. Too large a value will make the model more prone to overfitting.

In the third iteration, two more models were added and compared to the SVM. A default `LogisticRegression` implementation from scikit-learn was used as a baseline model. Further, the XGBoost implementation `XGBClassifier`, from the xgboost [7] library, was used with default parameters.

4.7 Grid search and hyperparameter tuning

In order to perform hyperparameter tuning, i.e. find the best hyperparameters for the estimator, as described in Section 2.4.6, `GridSearchCV` and `HalvingGridSearchCV` from scikit-learn [22] were used in the first iteration. The GridSearchCV uses the whole training set and tries all the different combinations of the parameters specified, and returns the optimal combination. In contrast, the HalvingGridSearchCV repeatedly applies small amounts of data and keeps training on the best parameters until the optimal is found. This improves the speed significantly compared to the GridSearchCV.

4.8 Evaluation

The models were evaluated using two different evaluation methods that we call the *row evaluation method* and the *first error evaluation method*. In the first two iterations, the row evaluation method was used, and in the third iteration both the first error evaluation method and the row evaluation method were used. In this section, both methods are explained.

4.8.1 Row evaluation method

With the row evaluation method, the models were evaluated based on how accurate they classified each row as either containing or not containing a relevant error description. In the first iteration, MCC and accuracy were used to compare the models. In the second iteration, MCC as well as F1-score, precision, recall, and accuracy were used to compare performance. The reason for adding more metrics was to get more detailed insight into the model's performance. Finally, in the third iteration, the row evaluation method was used on the manually annotated dataset, and the models were compared based on average R-precision, F1-score, and MCC. Average R-precision was added in order to evaluate how good the models were at finding the correct relevant error descriptions.

4.8.2 First error evaluation method

In the third iteration of experiments, the models were evaluated based on how accurate they could predict which row in each file that contained a relevant error description, in order to be able to compare their performance with the BFA's. The evaluation was carried out on a test set created from the first error dataset, which only contained one line with a relevant error description per file. For each file in the test set, a prediction was recorded that represented whether the model could predict a relevant error description on the correct line in the file or not. If the model only predicted one error description per file, and it was the correct line, a 1 was recorded. Otherwise, a 0 was registered for the entire file. To calculate the accuracy, the recordings were then compared to a list of 1:s, that represented the number of errors in each file.

4.9 Visualisation of predictions

In the third iteration of experiments, we generated visualisations of predictions from the SVM and XGBoost models in order to get an insight in how much each feature contributed to the prediction. The visualisations were generated by the tool Lime [25], that can be used for both text and image classification with classical machine learning models as well as neural networks. As input, it takes a prediction function and a log line, and the output shows how much each feature of the line contributed to the prediction.

Chapter 5
Result

This section starts with a summary of the interview with the case company. Thereafter, the results from the experiments of each iteration are presented.

5.1 Interview

According to the Senior Software Engineer at Axis our model will not outperform the BFA on its handcrafted regular expressions since they provide exact matches that are well tested. What is valuable for the case company is if the model can find any error that the BFA cannot find. Since the errors in the log files will change with the development of the build system, new errors will show up in the log files regularly. If the AI-model could help the engineers to navigate to any relevant error description in any case where the BFA cannot find anything today, it would save time and be helpful for them.

In order for the AI-model to be useful at the case company, it is important that the information from it is presented to the user in an intuitive way. According to the engineer it can be done through a Jenkins plugin. Then the user will be able to compare the information from the two plugins and decide in each case which one to rely on. In the cases where the BFA does not provide any help, it will be valuable for the user to get a hint from the AI-model of where the first error could be, and then there will be consequential errors following it. The engineer added that it is important that the analysis is done fast since the rows are generated in real time. If the data analysis part is written so that it is connected to Jenkins it will have direct access to the log files and can analyse them as they are generated.

Regarding future maintenance of the model, the engineer suggested either to train the model continuously, or to provide a service that can be activated manually to train the model when needed. In what way to train the model is however a question that needs to be investigated further and possibly discussed with engineers with more experience in AI. He also emphasised that regardless of exactly how the AI-model is trained, updating the model is important since changes in the build system are introduced regularly. It also happens that

new programming languages are introduced in the code, which leads to new types of errors appearing in the log files. In order to find out what would be the best way to train the model, data about the training needs to be collected. For example, it is important to find out how long time it takes to train it.

In summary, it can be stated that the most important requirements on the model are that: the model finds error descriptions the BFA is currently not able to find, it is fast enough to make predictions in real time during the build process, and it is easy for the engineers to maintain.

5.2 First iteration

In this section, results from the first iteration of experiments are presented. First, we explain how we selected the model and verified the pipeline. Thereafter, results from three experiments with different data split techniques, on the dataset annotated with the entire log method, are shown. The iteration is concluded with results from an experiment with hyperparameter tuning.

5.2.1 Pipeline verification

An initial pipeline with a TF-IDF vectoriser and an SVM model was assembled. To verify the pipeline, it was tested on a simple text classification dataset [17] with emails annotated as either *spam* or *ham*. This dataset was selected because it is a similar text classification problem to ours. The SVM achieved an accuracy of 0.98 and an MCC of 0.93.

5.2.2 Random split

Two experiments were carried out with randomly split data. The model used was the SVM and the number of 0:s were undersampled so they were as many as the 1:s. As seen in Table 5.1, the model was trained on both a dataset consisting of failed build logs from 15 and 61 days, and performed better on the larger dataset. However, due to the dataset being very large, as seen in Table 4.2, the total time for both the training and evaluation of the model was four days. Therefore, it was not feasible to conduct more experiments with that amount of data so all subsequent experiments were carried out with data from 15 days.

Table 5.1: Results obtained with the SVM trained on different dataset sizes

Model	Days	N-gram	Undersampling factor	Accuracy	MCC
SVM	15	(1,1)	1	0.99	0.59
SVM	61	(1,1)	1	0.99	0.71

5.2.3 Server stratification

One experiment was conducted using data stratified based on server. As described in Section 4.5.1, the thought behind splitting the data based on data source was that the data from different servers might look different. In this experiment, an undersampling factor of 0.5 was used, so more data was kept during training and rows containing relevant error descriptions made up a third of the data. The reason for changing the undersampling factor was that the amount of training data in this experiment decreased due to that there was almost a 50/50 ratio between the training and test sets. In this experiment, the model performed worse than when trained on randomly split data, with an accuracy of 0.99, and an MCC of 0.36. Further, the same regular expressions were still present in both the training and test datasets and therefore we decided to try another stratification strategy.

5.2.4 Regular expression stratification

The next experiments were carried out with a dataset stratified on regular expressions. As explained in Section 4.5.1, the stratification was done by randomly choosing 20% of the regular expressions present in the BFA for the test set in order to determine whether the model could predict new relevant error descriptions it had not been trained on. The models were trained with an undersampling factor of 1, i.e. an equal distribution of 1:s and 0:s. Initially, models trained on this data did not receive satisfying results. To try to improve the results, we decided to tackle the imbalanced regular expression distribution with regular expression oversampling. This modification improved the results significantly as seen in Table 5.2.

Table 5.2: Results with the SVM with and without regular expression oversampling

Model	Regular expression oversampling	Accuracy	MCC
SVM	no	0.99	0.21
SVM	yes	0.99	0.48

5.2.5 False positive example

An example of a false positive prediction by the SVM model trained with regular expression oversampling, with results presented in Table 5.2, is a frequently occurring line about memory leaks saying: `definitely lost 0 bytes in 0 blocks`. However, the regular expression the model was trained on is: `.*definitely lost: [1-9][0-9]* bytes .*`. According to the regular expression, the line is only considered a relevant error description if the line states that anything above 0 bytes is lost. Unfortunately, this difference is not captured by the model, which predicts relevant error descriptions on lines stating that no memory has been lost.

5.2.6 Hyperparameter tuning

Grid search was performed on both the TF-IDF vectoriser and the SVM classifier in order to find the best hyperparameters. The best parameters found with `HalvingGridSearchCV` are presented in Table 5.3.

Table 5.3: The optimal set of parameters for TF-IDF and SVM found with Grid search

Model	Parameter	Values	Best value
TF-IDF	min_df	1, 0.1, 0.01, 0.05, 0.001	1
	max_df	0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0	0.7
	ngram_range	(1,1), (2,2), (3,3), (1,2), (1,3), (2,3)	(1,3)
SVM	C	0.1, 1, 10, 100, 200	100
	gamma	1, 0.1, 0.01, 0.001	0.01

A model with the best values from hyperparameter tuning as input arguments was trained on data from 15 days undersampled with a factor of 1, and stratified on regular expressions. The model received an MCC score of 0.15, which is lower than the model trained with default parameters. Therefore, no more experiments with hyperparameter tuning were carried out.

5.3 Second iteration

After completing the experiments in the first iteration, it was discovered that the procedure for annotating the dataset was not optimal. The problem was that the BFA only matches one line per regular expression, e.g. a build log can contain many lines that match the same regular expression but only the first one is saved by the BFA. The lines following the first match are the so called consequential errors mentioned in the interview in Section 5.1. Since the annotation only uses the BFA's matched lines, only the first match for every regular expression was annotated with 1 in the dataset and the other lines that matched the same regular expression were annotated with 0. In order to avoid including those lines in the dataset, a new annotation method, the first error method, was created. The new approach was to omit all log lines after the first error is found, as described in Section 4.4.1. All remaining experiments in this study were conducted with models trained on the dataset annotated with the first error method.

5.3.1 Different undersampling strategies

The impact of different undersampling factors on the SVM was investigated on data from 15 days. As seen in Table 5.4, the lower the undersampling the better the model performed. Unfortunately it was not feasible to train the SVM without undersampling, since the dataset is approximately 17 million rows, as seen in Table 4.2, so the experiments were terminated earlier than desired. In the third iteration of experiments, an undersampling factor of 0.01 was used since it was not feasible to run many different models with an undersampling factor of lower than 0.01, since that made total time of training and evaluation increase from a couple of hours to around one day.

Table 5.4: Results with the SVM with different undersampling factors

Model	Undersampling factor	Accuracy	Precision	Recall	F1-score	MCC
SVM	1	1.0	0.07	0.99	0.13	0.26
SVM	0.1	1.0	0.34	0.99	0.51	0.58
SVM	0.01	1.0	0.65	0.98	0.78	0.80
SVM	0.001	1.0	0.99	0.97	0.98	0.99

5.4 Third iteration

After completing the experiments in the second iteration, it was discovered that the evaluation method used was not suitable to evaluate the performance on the first error dataset. Therefore, two new evaluation methods were introduced in the third iteration.

The first evaluation method, described in Section 4.8.2, is a modified version of the previously used row evaluation method. It had to be adapted to the new dataset since the dataset

now only includes the first relevant error description that the BFA has found in each file. The new evaluation method, the first error evaluation method, was used for the experiments that served as an evaluation of how good our model was at predicting relevant error descriptions it had been trained on. For the second evaluation in the third iteration, a manually annotated dataset was created, as described in Section 4.4.1. The evaluation on the manually annotated dataset was performed in order to find out if the model could predict new relevant error descriptions. Since the manually annotated dataset consisted of entire log files with multiple error descriptions in each file, row evaluation with F1-score and MCC was used. Further, average R-precision was calculated in order to evaluate how good the model was at finding the correct relevant error descriptions.

5.4.1 Baseline model

The experiments in the first and second iteration were conducted with only the SVM. To ensure that the SVM was a good model to use on our dataset, we included Logistic regression as a baseline model in the third iteration. The model was trained without undersampling, and achieved an accuracy of 0.87 as seen in Table 5.5.

Further, the default SVM model was trained and evaluated on data undersampled with a factor of 0.01.

Table 5.5: Comparison of Logistic regression and SVM

Model	Undersampling factor	Accuracy
Logistic regression	-	0.87
SVM	0.01	0.90

5.4.2 XGBoost

Three experiments with the XGBoost model were conducted in order to compare it to the SVM model. As the models trained on data undersampled with a factor of 0.01 did not predict any relevant error descriptions, we decided to increase the undersampling factor until the model started to predict relevant error descriptions. In the third experiment, the model started to predict some relevant error descriptions, and achieved an accuracy of 0.03. Since this result was not satisfactory, we moved on to try to improve the SVM model instead.

Table 5.6: Results from XGBoost trained with different undersampling factors

Model	Undersampling factor	Accuracy
XGBoost	0.01	0
XGBoost	0.1	0
XGBoost	1	0.03

5.4.3 Regular expression oversampling

A default version of the SVM was trained on data from 15 days with an undersampling factor of 0.01 and regular expression oversampling. The model achieved an accuracy of 0.18. Since this result was significantly lower than the default SVM without oversampling, no more experiments with oversampling were done.

5.4.4 Manual hyperparameter tuning

Hyperparameter tuning was carried out in the first iteration of experiments, but did not result in a better model, as seen in Section 5.2.6. In the third iteration, different values for the hyperparameters of TF-IDF: `ngram_range`, `max_df`, and `min_df`, described in Section 4.5.2, were tested manually.

Results from experiments with three different `max_df` values are presented in Table 5.7. The best performing models were the SVM models with `max_df` values of 0.5 and 0.75.

Table 5.7: Results with SVM with different `max_df` values

Model	max_df	Accuracy
SVM	1	0.90
SVM	0.25	0.91
SVM	0.5	0.93
SVM	0.75	0.93

In Table 5.8, results obtained with different values of `min_df` are presented. The best performing models were the SVM models with `min_df` values of 1 and 10. Both models achieved an accuracy of 0.90, which shows that changing `min_df` from the default value of 1 to 10 does not improve our model.

Table 5.8: Results with SVM with different `min_df` values

Model	min_df	Accuracy
SVM	1	0.90
SVM	10	0.90
SVM	50	0.88
SVM	100	0.84

Different `ngram_ranges` were evaluated, and the results are presented in Table 5.9. Adding either just bigrams or both bi- and trigrams improved the model compared to the default one which only uses unigrams.

Table 5.9: Results with SVM with different `ngram_ranges`

Model	N-gram range	Accuracy
SVM	(1,1)	0.90
SVM	(1,2)	0.92
SVM	(1,3)	0.92

5.4.5 Dimensionality reduction

The original number of features in the 15 days dataset with an undersampling factor of 0.01 was 30856, and truncations to different number of features were examined, as seen in Table 5.10. The results show that adding dimensionality reduction did not improve the model.

Table 5.10: Results with the SVM with different number of features with a TruncatedSVD

Model	Features	Accuracy
SVM	64	0.34
SVM	128	0.45
SVM	256	0.67
SVM	512	0.76

5.4.6 Unannotated relevant error descriptions

The first error dataset was created in order to avoid unannotated relevant error descriptions in the dataset. However, we discovered that the first error dataset also includes unannotated error descriptions. For example, in one build log, the line: `ERROR: Task X failed with exit code '1'` is not annotated as a relevant error description. However, the line: `Summary: There was 1 ERROR message shown, returning a non-zero exit code`, which occurs further down in the same log file, is annotated as a relevant error description. It is clear that the second line occurs as a consequence of the first. Nonetheless, no regular expression corresponding to the first line is present in the BFA, and therefore it is not annotated even though it contains a relevant error description. Thus, a manually annotated dataset, described in Section 4.4.1, was created for the final evaluation step.

5.4.7 Evaluation on manually annotated dataset

Results from the default versions of the SVM and XGBoost evaluated on the manual dataset are presented in Table 5.11. The SVM and XGBoost were trained on their best performing undersampling factor, 0.001 and 1 respectively, from previous experiments as seen in Table 5.4 and Table 5.6.

Table 5.11: Results from the SVM and XGBoost models on the manually annotated dataset

Model	Undersampling factor	Average R-precision	F1-score	MCC
SVM	0.001	0.25	0.50	0
XGBoost	1	0.26	0.57	0.17

Since the SVM model did not predict any relevant error descriptions correctly in the manually annotated dataset, we decided to experiment with different undersampling factors again. Out of the tested undersampling factors, the optimal value for SVM on the manually annotated dataset proved to be 1, as seen in Table 5.12.

Table 5.12: Results from default SVM on manually annotated dataset with different undersampling factors

Model	Undersampling factor	Average R-precision	F1-score	MCC
SVM	1	0.19	0.60	0.20
SVM	0.1	0.14	0.50	0
SVM	0.01	0.21	0.50	0
SVM	0.001	0.25	0.50	0

We know that the model can predict relevant error descriptions it has already seen with an F1-score of 0.93, but performed significantly worse on new error descriptions and therefore we decided to try to generalise the models by adding a TruncatedSVD. In Table 5.14 and Table 5.13, results from different experiments with the XGBoost and SVM with different number of features are presented. Both models were trained with an undersampling factor of 1.

Table 5.13: SVM with different number of features

Model	Features	Average R-precision	F1-score	MCC
SVM	64	0.14	0.59	0.18
SVM	128	0.16	0.58	0.16
SVM	256	0.16	0.58	0.17
SVM	512	0.17	0.56	0.13

Neither the SVM nor the XGBoost model performed better when TruncatedSVD was added. The best performing XGBoost model was the one presented in Table 5.11, with an average R-precision of 0.26, an F1-score of 0.57, and an MCC of 0.17. The model trained on data undersampled with a factor of 1 was the SVM model that overall achieved the best scores with an average R-precision of 0.19, an F1-score 0.60 and an MCC of 0.20, as seen in Table 5.12. Generally, both models scored low values of both average R-precision, MCC, and F1-score.

Table 5.14: XGBoost with different number of features

Model	Features	Average R-precision	F1-score	MCC
XGBoost	64	0.25	0.49	0.10
XGBoost	128	0.29	0.50	0.08
XGBoost	256	0.29	0.48	0.08
XGBoost	512	0.22	0.50	0.06

The confusion matrices for the respective best models are presented in Table 5.16 and Table 5.15. The SVM model predicted 129 relevant error descriptions out of 646, and incorrectly predicted 427 rows as 1:s, as seen in Table 5.15.

Table 5.15: Confusion matrix for the SVM model

		Predicted class		
		0	**1**	**Total**
Actual class	**0**	60 050	427	**60 447**
	1	517	129	**646**
	Total	**60 567**	**556**	

Table 5.16 shows that the XGBoost model predicted 192 correct relevant error descriptions out of 646, and incorrectly predicted 1508 rows as relevant error descriptions.

Table 5.16: Confusion matrix for the XGBoost model

		Predicted class		
		0	**1**	**Total**
Actual class	**0**	58 969	1 508	**60 447**
	1	454	192	**646**
	Total	**59 423**	**1 700**	

5.4.8 Visualisation of predictions

In this section, two examples of visualisations of predictions created with the Lime tool are presented. The first example, in Figure 5.1, was created with the best performing XGBoost model and the second example, in Figure 5.2, was created with the best performing SVM model. Both visualisations consist of three parts: to the left the probability that the model will predict either class 0 or class 1 is shown, to the right the six most relevant features from the row and their respective weights are listed, and down below the row, together with its most relevant features highlighted is displayed. In contrast to the SVM implementation, the XGBoost implementation only makes it possible to extract probabilities for features that contribute to a prediction of class 1, thus, for that model only tokens contributing to the class 1 prediction can be visualised.

In Figure 5.1 below, the example of a prediction from the best performing XGBoost model is shown. In this example, the probability that the model predicts a relevant error description is 0.99. However, it is not annotated as a relevant error description by the engineers since it is a warning. The feature that contributed most to the prediction of class 1 is the word *failed*. The weights of the features can be interpreted by subtracting them from the prediction probabilities. For example, if we remove the word *failed* from the text visualised in Figure 5.1, we expect the classifier to predict class 1 with a probability of 0.99 - 0.97 = 0.02 instead.

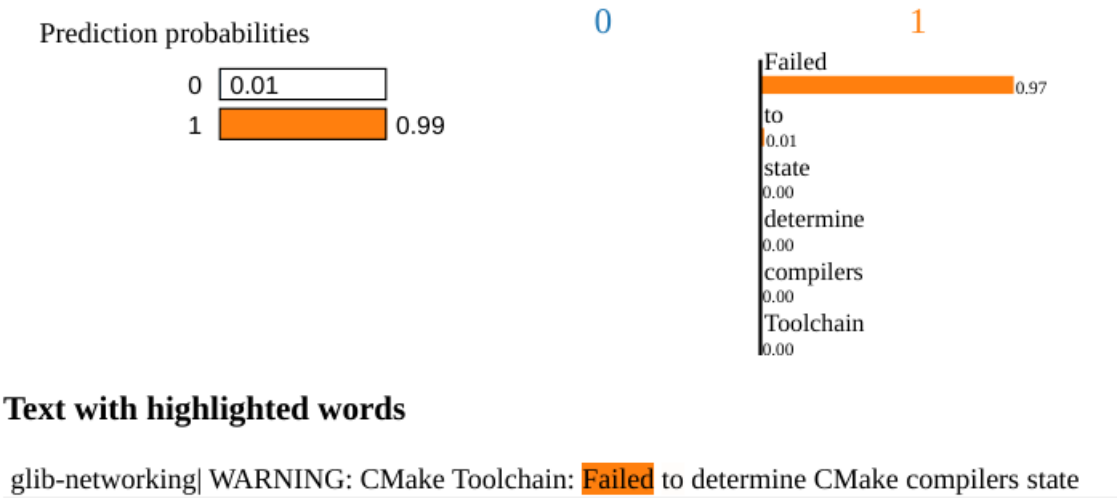

Figure 5.1: Visualisation of an incorrect prediction

The example of the prediction by the best performing SVM model is shown in Figure 5.2. In the example, the probability that the model predicts a relevant error description is 0.97. The example is a line that does not contain any corresponding regular expression in the BFA. The features that contributed most to the prediction are the words *failed* and *error*. However, we can also see that the words *for* and *to* contributed to the prediction. Lastly, we can see that the number *99* contributed to the prediction probabilities.

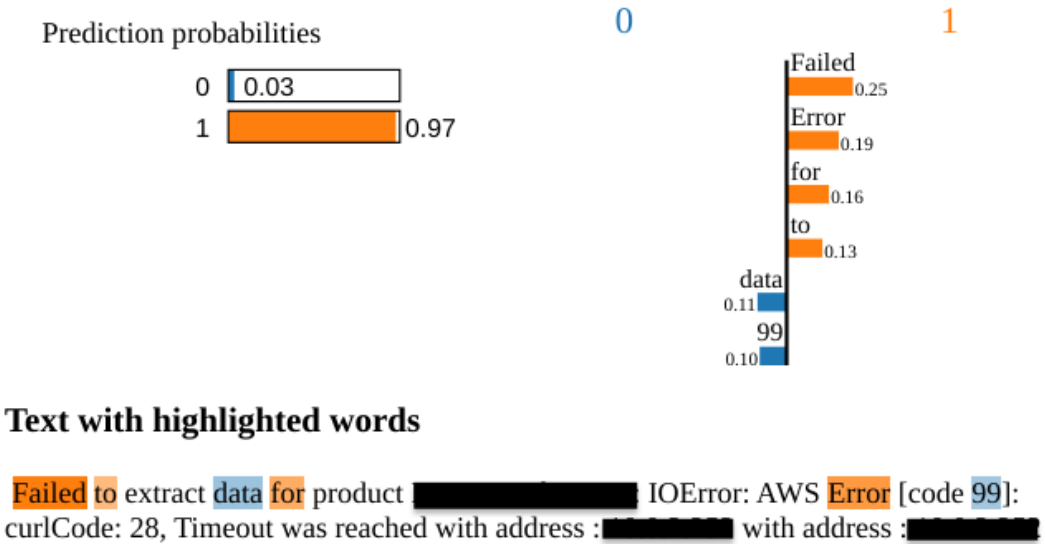

Figure 5.2: Visualisation of a correct prediction

Chapter 6
Discussion

6.1 First iteration

In this section, the results from the first iteration of experiments are discussed. The problem with the annotation of the dataset are explained as well as the undersampling factors and metrics used. Further, the three stratification strategies and their purpose are described and evaluated.

6.1.1 Dataset

All experiments in the first iteration were conducted using the entire log dataset, described in Section 4.4.1, which contained all rows from build logs from 15 days. There was, however, an issue with that dataset. Since the BFA only registers the first matching line of each regular expression pattern, there are many lines in the log files that contain relevant error descriptions, but are annotated with 0:s when annotating the log files solely with data from the BFA. This implies that when training on the entire log dataset, the model is trained on lines with similar text and different `is_faulty` annotations. In order to avoid including ambiguous lines in the data, a different method for creating the dataset, the first error annotation method, was used in the second and third iteration.

6.1.2 Undersampling

Two different values of the `sampling_strategy` parameter were used: 1 and 0.5. A value of 1 means that undersampling is performed until there is an even distribution between the two classes. The intention of this sampling strategy was to achieve an even distribution in order to avoid getting a biased model. However, as seen in Table 4.2, to achieve an even distribution 19 million rows are undersampled to approximately 7700 rows with 1:s and the equal amount

of rows with 0:s. This implies that a lot of data is lost. Therefore, different undersampling strategies were evaluated in the second iteration.

6.1.3 Metrics

Accuracy and MCC were the only metrics calculated in the first iteration of experiments. As seen in Table 5.1 and 5.2, all models tested received an accuracy of 0.99. Since the dataset used for evaluation consisted of almost only 0:s, even a model that only predicts 0:s would have achieved a high accuracy. However, we were more interested in knowing if the model could predict 1:s and therefore accuracy could not be considered a suitable measurement, since a high accuracy can be achieved by a model that only predicts 0:s. MCC is better for measuring performance on imbalanced datasets, since it gives equal importance to classifications of both classes. Even though undersampling was used to balance the training set, the evaluation set was kept imbalanced, and therefore MCC can be considered a suitable metric for evaluation. Since we only recorded two metrics, we did not get detailed insight in the behaviour of the model. Therefore, precision, recall, and F1-score were added in the second iteration.

6.1.4 Stratification strategies

Since the model was supposed to complement and not replace the BFA at the case company, in this iteration we focused on measuring the models' ability to predict relevant error descriptions it had not already been trained on. This was done through experimenting with different stratification strategies.

Initially, the dataset was split randomly without stratification and the results in Table 5.1 show that the model was able to predict relevant error descriptions. However, since the split was done randomly, some regular expressions could have been present in both the training set and the test set. Therefore, we could not determine whether the model could predict any relevant error descriptions it had not already been trained on.

The experiments performed with data stratified based on server resulted in an MCC score much lower than the previous experiments, 0.36 compared to 0.59, as presented in Section 5.2.3. Splitting the data based on the server resulted in almost a 50/50 ratio between the training and test sets, due to the distribution of data between servers, shown in Table 4.1. Therefore, the training set was smaller than in the experiments done with a random split. To compensate for the smaller training set, the undersampling factor was lowered to 0.5. However, since the undersampling factor only affects the number of 0:s in the dataset, a 50/50 distribution still resulted in a lower number of 1:s. This could have been the cause of the low MCC score. Further, with this stratification strategy we could still not confirm that the model could predict on regular expressions that it had not been trained on, since there were regular expressions present in both the training data and regular expressions present in the test data.

The initial experiment with regular expression stratification resulted in an MCC score of 0.21, as shown in Table 5.2. When adding regular expression oversampling, the MCC score increased significantly from 0.21 to 0.48. However, the method used to stratify the data based on regular expressions was flawed and therefore the results cannot be considered valid. When choosing which regular expression to put in the training and test data respectively, we started from a list of all regular expressions present in the BFA. Since that list consisted of many

more regular expressions than those actually present in the data, one cannot assume that the regular expressions picked for the test set were actually present in the data. Therefore, it is not guaranteed that stratification was actually performed. Further, choosing 20% of the regular expressions randomly is probably not an optimal approach since the distribution between regular expressions in the data is uneven. A better approach could have been to use cross-validation, i.e. to run several iterations with different selections of regular expressions.

Before evaluating whether the model can predict relevant error descriptions it has not been trained on, it is important to assure that it can predict relevant error descriptions it has been trained on. Therefore, the experiments in the second and third iteration were conducted using random split, until the best model was found. After that, a manually annotated dataset was used to determine whether the model could predict relevant error descriptions it had not been trained on.

6.1.5 Hyperparameter tuning

In the first iteration we tried to tune the hyperparameters of TF-IDF and SVM using HalvingGridSearch. However, the tuned model performed significantly worse than the model with default parameters. Undersampling with a factor of 1 combined with regular expression oversampling made the proportion of 1:s and 0:s in the training dataset balanced, while the test dataset remained imbalanced. Since only the training dataset was used to tune the hyperparameters, a possible reason for the bad performance could be that the model was optimised for a dataset with a 50/50 ratio between 1:s and 0:s and evaluated on an imbalanced dataset. Because of this, we had to try different configurations of the hyperparameters manually instead.

6.2 Second iteration

For the experiments in the second iteration, a new dataset was created with the first error annotation method, as described in Section 4.4.1. An SVM model was trained with different undersampling strategies to evaluate the impact of changing the ratio of 1:s and 0:s in the dataset. In this section, the results from those experiments will be discussed.

6.2.1 Undersampling

In Table 5.4, results from experiments with four different undersampling factors are presented. The lowest undersampling factor that we tested was 0.001, which also received the best performance with an MCC of 0.99 and an F1-score of 0.98. The more undersampling, the more prone the model become to predicting relevant error descriptions. Therefore, the undersampling factor affected the precision metric, which decreased from 0.99 to 0.07 when the undersampling factor increased from 0.001 to 1. As seen in Table 4.2, the evaluation dataset is imbalanced with only 0.001% 1:s. One possible explanation could be that when the model was trained on data with a larger proportion of 1:s than in the evaluation dataset, it became more inclined towards predicting 1:s.

6.2.2 Evaluation method

As seen in Table 5.4, the best model trained in this iteration achieved higher MCC score than the best model from the previous iteration, 0.99 compared to 0.59 in the first iteration. The reason for this increase could be that the wrongly annotated lines were omitted in the first error dataset. However, since the models were still evaluated using the row evaluation method, the results only show how good the models were at predicting if a single line contained a relevant error description or not.

In order to compare our model to the BFA, we needed to change evaluation method. Since the BFA finds the first line of each matching regex in each log, we implemented a new evaluation method that evaluated how good the model was at finding the first relevant error description on the correct row in each file, as described in Section 4.8.2.

6.3 Third iteration

In the third iteration of experiments, the models were trained on the first error dataset, and evaluated in two different ways. First, the models were evaluated with the first error evaluation method in order to compare our model to the BFA. Then, the models were evaluated on a manually annotated dataset to find out whether they could predict relevant error descriptions they had not been trained on.

6.3.1 Evaluation on first error dataset

The purpose of the experiments on the first error dataset was to determine if the model could learn the relevant error descriptions that it had been trained on. In the previous iterations, we immediately tried to create a model that would complement the BFA. However, it is not realistic to assume that the model can complement the BFA and predict unseen error descriptions before it can predict error descriptions it has been trained on.

The highest accuracy achieved by the SVM model on the first error dataset was 0.93. In the second iteration, the results showed that more undersampling made the model perform worse. Additionally, in the third iteration the regular expression oversampling proved to significantly worsen the performance. Both of these modifications result in a training set where either the distribution of 1:s and 0:s or the distribution of regular expressions differ from the evaluation set. This could indicate that the model only learns the exact relevant error descriptions and the distribution.

Even though the model proved to be able to predict relevant error descriptions, the experiments on the first error dataset cannot be used to determine if the model can predict relevant error descriptions it has not been trained on. Therefore, experiments on a manually annotated dataset had to be done.

6.3.2 Evaluation on manually annotated dataset

The results show that the model can predict relevant error descriptions that are not present in the BFA. However, the performance on the manually annotated dataset was significantly lower than on the first error dataset. The difference in performance indicates that the model

is overfitted to the training data. When looking at the confusion matrices in Table 5.15 and Table 5.16, one can see that both models predict more relevant error descriptions than what are present in the evaluation set. This could be due to that the models are trained on data with an even distribution of 1:s and 0:s, but evaluated on imbalanced data with a ratio of 1:100 between 1:s and 0:s. Lowering the undersampling factor had proved to be successful previously, but with the manually annotated dataset, lower undersampling factors resulted in models not predicting any row as a relevant error description. Although the SVM predicted less false positives than the XGBoost, the number of false positives is still higher than the number of true positives. Thus, it is more likely that a class 1 prediction is incorrect than correct, which leaves a lot of responsibility to the engineer who in that case must decide for themselves whether it was a correct prediction.

6.3.3 Visualisation of predictions

In the first prediction visualised in Figure 5.1, the main feature that contributed to the model predicting the row as a relevant error description was the word *failed*. This is reasonable since one can assume that a row that contains the word *failed* most likely contains information about an error. However, the model misses the crucial feature *warning*, which makes us not want to mark the row as a relevant error description. The dataset the model was trained on contains seven different regular expressions with the word *failed*, whereof one also contains the word *warning*, namely the regular expression: `WARNING: Setscene task .*(meta.*)(, |:)(do_.*)\\) failed with exit code '1' - real task will be run instead`. It is therefore reasonable that the model predicts rows with the word *failed* as relevant error descriptions, even though they contain the word *warning*. However, in the manually annotated dataset this warning was not marked as a relevant error description. The fact that some warnings are considered relevant error descriptions and others not makes it very difficult for a general model to predict relevant error descriptions correctly.

In the second example in Figure 5.2, the four features that contributed the most to the prediction of class 1 were *failed*, *error*, *for*, and *to*. It is reasonable that the features *failed* and *error* contribute to a prediction of class 1. However, *for* and *to* are words that do not contribute to the meaning of the text and therefore they should have been filtered out as stopwords. We used a value of 0.5 for `max_df` to filter out words, which means that we want to filter out features that occur in more than 50% of the rows, but as we can see, a good idea would have been to combine `max_df` with an English stopword list. The reason for not using an English stopword list is that the build logs do not really contain natural language. Further, the number 99 contributed to the prediction. This could have been prevented by filtering out numbers from the log lines.

6.4 Data split

In the experiments where random split was used, the data was split with an 80/20 ratio between training and test data. Since the dataset for 15 days consisted of around 19 million rows, as shown in Table 4.2, the split resulted in a test set of almost 4 million rows. The magnitude of the test set made the evaluation time-consuming, so another ratio between the test set and training could have been more suitable for this dataset. That would not only speed up the evaluation time but also give the model more data to train on.

6.5 Practical usage at the case company

In the interview it emerged that the model is intended to serve as a complement to the BFA. It is valuable for the engineers if the model can find any relevant error description in a file where the BFA did not. The new error descriptions, that are found by the AI-model, can then be used to help the engineers create new regular expressions that can be added to the BFA. As the BFA is updated with more regular expressions, more relevant error descriptions will be caught. The model can then be retrained with a new dataset and eventually in the future it might even be able to completely replace the BFA.

6.6 Threats to validity

The largest threats to the validity of the results appear in the dataset annotation step. For example, it was not possible to verify that all files had been annotated correctly, since the dataset consisted of approximately 19 million rows from 1 757 logs, as seen in Table 4.2. We tried to tackle this issue through manually verifying the annotation on a smaller dataset, but since there are many different regular expressions we can still not be certain that the dataset is correct in a larger scale. For example, when implementing the method for creating the first error dataset, it was noted that for one regular expression, the encoding of the files caused a mismatch between the `matchingString`, and the log line it was supposed to match. Even though this particular problem was fixed by matching on `pattern` instead, there might have been more errors in the dataset that were not discovered due to how difficult it was to verify the annotations. Further, the regular expressions used by the BFA are handcrafted, and cannot be guaranteed to be 100% correct and include all error descriptions they are thought to include. We used a manually annotated dataset in the final evaluation step since we wanted to minimise the number of incorrectly annotated lines. However, it is possible that it would have been better to use manually annotated data for training as well. With manually annotated data, the problems with the automatic annotation methods are avoided, but there is still a risk of wrongly annotated lines caused by human errors.

The build logs used to train the models were from builds run during 15 days. Due to time limitations we did not look into whether the builds that ran during those days were representative of builds that are commonly run at the case company's servers. It would have been interesting to look at what different types of builds are represented in the data and with that information possibly create a dataset with a larger variation of builds in order to create a more generalised model.

Since the training of the models is a stochastic process, there might be a slight variation of the performance of the models from different iterations. Therefore, it could have been more suitable to train each model a couple of times and then take an average of the performance to get a more reliable measurement.

The interview was conducted with a single engineer from the case company. It is possible that the engineer's opinions on the requirements of the model are not representative of all Jenkins users at the case company.

Chapter 7

Conclusion and future work

In this section, we answer the research questions and give suggestions on future work.

7.1 Conclusion

The results show that it is possible to detect new relevant error descriptions with an AI-model that has been trained on a dataset annotated solely with historical data from the BFA. However, the results for both the XGBoost and the SVM model evaluated on the manually annotated dataset were unsatisfactory. The XGBoost achieved an average R-precision of 0.26, an F1-score of 0.57, and an MCC of 0.17. SVM achieved an R-precision of 0.19, an F1-score of 0.60, and an MCC of 0.20. Since the SVM model performed much better when evaluated on data with relevant error descriptions it had already been trained on, achieving an accuracy of 0.93, it is possible that the model is overfitted. The evaluation on the manually annotated test set showed that the SVM model predicted approximately three times as many false positives as true positives, and XGBoost predicted almost eight times as many false positives as true positives. Consequently, using the model as a complement to the BFA would still require the engineers to manually determine which of rows that the AI-model predicted that actually contain relevant error descriptions.

The most important requirement on the model to be useful at the case company is that it can function as a complement to the BFA. In other words, if the BFA does not find any relevant error descriptions in a failed build log from Jenkins, any correct prediction by the model is useful for the engineer. However, our best model is probably more confusing than helpful for the engineer since it predicts far too many irrelevant rows as relevant error description.

7.2 Future work

In this section, possible improvements in different parts of the pipeline are suggested. First, improvements of the build logs and the dataset are described. Thereafter, improvements in the preprocessing and training step are proposed.

7.2.1 Dataset

Considering that the models trained on the first error dataset did not achieve high performance on the manually annotated dataset, it could be worth trying to train models on a differently annotated dataset. The BFA only saves the first line matching each regular expression in each build log, so to prevent that lines matching the same regular expression are annotated differently, each regular expression could be used to annotate all lines in the log that match it as a relevant error description.

Another way to improve the dataset could be to modify what is written to the build log files. In Section 5.2.5 it is noticed that the model is not able to distinguish between different numbers. For example, the line `definitely lost 64 bytes in 8 blocks` is annotated as a relevant error description in the training set, but the model predicts the line `definitely lost 0 bytes in 0 blocks` as class 1 as well, although only the first line indicates a memory leak. A possible solution could be to add *ERROR: Memory leak* in the beginning of a line when there is a memory leak. That way, the model is more likely to classify the line as a relevant error description since we have seen in the visualisations that the model recognises that a line containing the word *error* most likely should be classified as class 1.

A problem with only annotating the training dataset with the historical data from the BFA is that there might be lines containing relevant error descriptions not present in the BFA that are annotated with 0. If the model is then evaluated on the same new relevant error descriptions, it has learnt that they are not relevant error descriptions. A manually annotated training set would prevent that the model gets wrongly trained. Although manual annotation is time consuming, it would most likely result in a better model.

Additionally, a column representing what type of error the relevant error description belongs to, such as *infrastructure error* or *user error* could be added to the dataset. That way, a machine learning model could be trained to not only detect but also classify the relevant error descriptions. Classification would be useful for the engineers since it gives them further insight in what caused the error.

7.2.2 Preprocessing

Since the SVM model performed much better when evaluated on relevant error descriptions it had already been trained on, compared to the manually annotated dataset, it was concluded that the model might have been overfitted to the data. To prevent overfitting, it could be a good idea to introduce more variation in the dataset. For example, duplicate lines could be removed and the most frequently occurring regular expressions could be undersampled.

The visualisation of the predictions showed that there are irrelevant words that contribute to the predictions. To filter out such words, a domain specific stopword list could be

created and used in combination with `max_df` and English stopwords.

7.2.3 ML-model

We intended to try a deep learning model, but it was not possible to include in this study due to time limitations. As previously mentioned, another study by Studiawan et. al [27] has shown better results using deep learning models on a similar task. Therefore, it would be interesting to see if a deep learning model would perform better on our dataset as well.

References

[1] Axis Communications AB. About Axis. Available from: `https://www.axis.com/about-axis`, 2022. Accessed: 2022-05-03.

[2] Steven M. Beitzel, Eric C. Jensen, and Ophir Frieder. Average R-Precision. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*. Springer US, Boston, MA, 2009.

[3] Hoss Belyadi and Alireza Haghighat. *Machine Learning Guide for Oil and Gas Using Python*, page 186. Gulf Professional Publishing, 2021.

[4] Carolin E. Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. Logchunks: A data set for build log analysis. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pages 583–587, 2020.

[5] Build Failure Analyzer contributors. Build Failure Analyzer. Available from: `https://plugins.jenkins.io/build-failure-analyzer/`, 2022. Accessed: 2022-04-26.

[6] Armin Catovic, Carolyn Cartwright, Yasmin Tesfaldet Gebreyesus, and Simone Ferlin. Linnaeus: A highly reusable and adaptable ML based log classification pipeline. In *Proceedings of the 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, pages 11–18, 2021.

[7] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[8] David Chicco, Niklas Tötsch, and Giuseppe Jurman. The Matthews correlation coefficient (MCC) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation. *BioData Mining*, 14(1):13, 2021.

[9] Adam Debbiche, Mikael Dienér, and Richard Berntsson Svensson. Challenges when adopting continuous integration: A case study. In *Proceedings of the International Confer-*

ence on Product-Focused Software Process Improvement (PROFES 2014), volume 8892, pages 17 – 32, 2014.

[10] Elasticsearch B.V. Elastic. Available from: `https://www.elastic.co/`, 2022. Accessed: 2022-04-25.

[11] Martin Höst, Björn Regnell, and Per Runesson. *Att genomföra ett examensarbete*. Studentlitteratur, 2006.

[12] Jenkins contributors. Jenkins. Available from: `https://www.jenkins.io/`, 2022. Accessed: 2022-02-08.

[13] Rahul Kumar. *Machine Learning Quick Reference*. Packt Publishing Limited, 2019.

[14] Brett Lanz. *Machine Learning with R, Second Edition*. Packt Publishing Limited, 2015.

[15] Scikit learn developers. Tuning the hyper-parameters of an estimator. Available from: `https://scikit-learn.org/stable/modules/grid_search.html`, 2007-2022. Accessed: 2022-04-12.

[16] Guillaume Lemaitre, Fernando Nogueira, and Christos K. Aridas. Imbalanced-learn: A Python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5, 2017.

[17] Mat Leonard. Spam dataset. Available from: `https://www.kaggle.com/datasets/matleonard/nlp-course?select=spam.csv`, 2022. Accessed: 2022-04-08.

[18] Didrik Lindqvist. Detection of infrastructure anomalies in build logs using machine learning. Master's thesis, Umeå University, umu-164730, 2019.

[19] Lakshmi Geethanjali Mandagondi. Anomaly detection in log files using machine learning techniques. Master's thesis, Blekinge Institute of Technology, 2021. 21179.

[20] Rafael G. Mantovani, André L.D. Rossi, Edesio Alcobaça, Joaquin Vanschoren, and André C.P.L.F. de Carvalho. A meta-learning recommender system for hyperparameter tuning: Predicting when tuning improves SVM classifiers. *Information Sciences*, 501:193–221, 2019.

[21] Pierre Nugues. *Language Processing with Perl and Prolog*. Springer, 2014.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] Ronaldo Prati, Gustavo Batista, and Maria-Carolina Monard. Data mining with imbalanced class distributions: Concepts and methods. pages 359–376, 01 2009.

[24] Max Rehkopf. What is continuous integration? Available from: `https://www.atlassian.com/continuous-delivery/continuous-integration`. Accessed: 2022-01-21.

[25] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should I trust you?: Explaining the predictions of any classifier. *CoRR*, abs/1602.04938, 2016.

[26] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning*. Cambridge University Press, 2014.

[27] Hudan Studiawan, Ferdous Sohel, and Christian Payne. Anomaly detection in operating system logs with deep learning-based sentiment analysis. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2136–2148, 2021.

[28] Wikipedia contributors. TF-IDF — Wikipedia, the free encyclopedia. Available from: `https://en.wikipedia.org/wiki/Tf%E2%80%93idf`, 2022. Accessed: 2022-02-04.

Appendices

Appendix A
Interview with Axis Communications AB

1. How accurate does the model need to be, does it have to outperform the BFA in order to be useful for Axis? How much room for errors are there?
2. What is required for the model to be added to your pipeline? Does it need to be a Jenkins plugin?
3. How do we minimise the maintenance of the model in the future? How do you make sure it is trained, who does it and how often is it done?

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-08-25

EXAMENSARBETE Identification of relevant error descriptions in build logs using machine learning
STUDENTER Lykke Axlin, Klara Broman
HANDLEDARE Martin Höst (LTH), Marcus Klang (LTH), Gustaf Lundh (Axis), Ola Söder (Axis)
EXAMINATOR Elin Anna Topp (LTH)

Utilizing artificial intelligence to help troubleshoot failed software builds

POPULÄRVETENSKAPLIG SAMMANFATTNING **Lykke Axlin, Klara Broman**

When software builds fail, software engineers often need to manually go through large unstructured log files to troubleshoot the failure cause. In our thesis, we investigate the possibility of using machine learning to automatically identify the relevant error descriptions in the log files.

Have you ever wondered how certain emails end up in your spam folder? The technique used to determine whether an email is spam or not is called text classification. In this study, we used text classification of rows in log files to help engineers find relevant error descriptions.

Many companies use the open source automation server Jenkins to facilitate continuous integration of new code in large code bases. When a build is run on Jenkins, a build log is created which contains information about the build, e.g. whether a test succeeded or not. The log files can sometimes become very large, and going through log files in order to troubleshoot why a build failed is a tedious task for an engineer. At Axis Communications, an open source plugin called the *Build Failure Analyzer* (BFA) is used to assist the engineers with this task. The BFA scans failed build logs for certain patterns in order to identify relevant error descriptions. These patterns are handcrafted by the engineers and new patterns need to be added continuously as new types of errors appear in the builds.

In our Master's thesis we investigated the possibility of using a machine learning model to automatically identify relevant error descriptions in the Jenkins build logs. The overall approach was to train a model on a dataset annotated with the findings from the BFA, and then evaluate if the model could detect error descriptions the BFA had not been able to detect. In the cases when the BFA does not find any relevant error descriptions in the build log, it would be very helpful if the AI-model could point the engineer in the right direction.

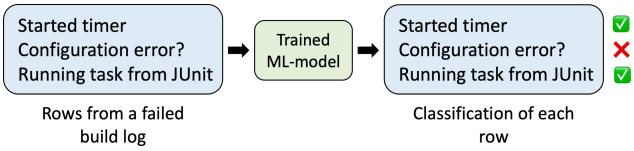

Figure 1: Prediction by a trained model on three build log lines

The results show that it is possible to train an AI-model to predict new relevant error descriptions in failed log files. However, the model predicts many more lines as relevant error descriptions than what are present in the file. This means that the engineer still needs to look through many log lines manually to find the relevant error descriptions. Therefore, further improvements to the model, as well as the dataset, are needed in order for the model to be useful in the industry.