DETECTION of INFRASTRUCTURE ANOMALIES in BUILD LOGS USING MACHINE LEARNING

Text classification on Continuous Integration log files.

Didrik Lindqvist

Didrik Lindqvist
Spring 2019
Masters Thesis, 30 ECTS
Supervisor: Adam Dahlgren Lindström
External supervisor: Henrique Truta
Examiner: Henrik Björklund
Master of Science in Computing Science and Engineering, 300 ECTS
Abstract

Continuous integration is a practice where software developers integrate their code to a bigger codebase multiple times per day. Before the integration, the code is built and tested by e.g. open source build tools such as Jenkins, and the information produced during this process is stored in a log file. Sometimes these builds fail, and the cause can be either user or infrastructure related. A user related error may be that the code cannot compile due to syntax error and an infrastructure error could be a DNS problem. This thesis evaluated how well machine learning can be used to label the cause on failed build logs as either user or infrastructure. This thesis compared the performance of three machine learning algorithms: support-vector machine, random forest, and gradient boosting classifier. Two different datasets are used in this study. A balanced dataset used for training and validation and another dataset used for testing. The preprocessing step, including feature selection, is done using term frequency-inverse document frequency, which converts the text from the build log to a machine learning friendly format. The study also evaluated three different sizes of n-grams for each algorithm and dataset. The performance for the three machine learning algorithms is evaluated by comparing the precision, recall, and F1-score for each model. The three machine learning algorithms and the methodology around preprocessing and evaluation are explained in this study. The results show that machine learning can be used as a tool to help the CI-owners, but may not be used to fully replace the classification done manually today. The machine learning algorithm that performed the best was gradient boosting classifier with an bag of 1 and 2-grams, with a precision, recall and F1-score of 0.87, 0.73 and 0.79.
Acknowledgements

I firstly want to thank my supervisor at the University, Adam Dahlgren, who have been helping me during the study with both technical and friendly support. I also want to thank my supervisor at Spotify Henrique Truta, Henrique have always been ready to chat or explain how the internal systems at Spotify work and have just been a joy to work with. I want to thank my girlfriend, Madeleine Gauffin, who has been a great support when times have been tough and energy low. I lastly want to thank all the wonderful people at Spotify and my friends that have supported me during this thesis.
Contents

1 Introduction 3
1.1 Problem specification 3
1.2 Outline 4

2 Theory 5
2.1 Background 5
2.1.1 Continuous Integration 5
2.1.2 Text classification 6
2.2 Preprocessing 6
2.2.1 Term frequency and inverse document frequency 6
2.2.2 N-grams 6
2.2.3 Stop words and minimum occurrence 7
2.3 Machine learning models 7
2.3.1 Support vector machine 8
2.3.2 Decision trees 9
2.3.3 Ensemble models 9
2.3.4 Random forest 10
2.3.5 Gradient boosting 10
2.4 Evaluation 10
2.4.1 Hyper-parameter optimization 12

3 Method 13
3.1 Dataset 14
3.1.1 Splitting on groups example 14
3.1.2 Preprocessing 15
3.2 Training models 16
3.3 Evaluating model 17

4 Result 19
4.1 Result from validation dataset 19
4.1.1 Support-vector machine 19
4.1.2 Random forest 20
4.1.3 Gradient boosting classifier 21
4.2 Result from test dataset 22
4.2.1 Support-vector machine 23
4.2.2 Random forest 24
4.2.3 Gradient boosting classifier 25
4.3 Classification on unknown build logs 26

5 Discussion 27
5.1 Conclusions 27
5.2 Future work 28
Contents

List of acronyms

CI - Continuous Integration

ML - Machine learning

TF-IDF - Term frequency–inverse document frequency

RF - Random forest

GBC - Gradient boosting classifier

SVM - Support vector machine
1 Introduction

At Spotify, there is a large number of people working on different software projects each day. These software projects vary in sizes, programming languages used, the number of contributors, and more. When developers start to contribute to the same software projects, it can create integration errors, and it is here Continuous integration (CI) comes in. One good description of CI comes from Martin Flower who said:

"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.” [1]

In the CI-systems at Spotify, multiple projects across different domains are compiled and built on a daily basis by a building tool called Jenkins, an open source tool for building and testing software [2]. The majority of the builds succeed, but a significant number of builds fail for various reasons. An interesting and important problem here is to understand what caused the build to fail. The cause of failure can be due to problems like infrastructure failures like Git-problems, timeouts, DNS issues, or user-related problems like syntax, code style, compile errors. It is important for the CI-owners to know this information, because they want to minimize the total number of infrastructure errors, so that the developers can focus on developing new software and have trust in the CI-system. These insights are currently gathered by manually creating regular expressions (regex) based on already known infrastructure and user errors. This regex method is not unique for Spotify, there are for example popular plugins for the build tool Jenkins that use the same approach [3]. However, it is not a reliable solution and does not work well with unseen errors. This is because the unseen errors most likely will not be found by the regex and the time it takes to manually inspect and create a new regex for each unknown build failure is too time-consuming.

A successful categorization of both infrastructure and user related errors will give the CI-system administrators a better estimate on how well the CI-system both perform and its reliability. The current solution does not allow the owners of the CI-system to manually investigate each build log generated by the CI-system on failure due to it being too time-consuming. Given the nature and structure of the build logs, a machine learning (ML) approach would be an excellent way to save both time and money for Spotify.

1.1 Problem specification

Gaining a better understanding of the performance of the CI-system makes the owner more aware of the platform reliability. This will result in less time troubleshooting for both the people managing the CI-system and those depending on the system to build their projects. A machine learning approach could remove the need to manually investigate build logs, leading to the CI-system owners getting more time to improve the system instead of analyzing it.
This thesis will evaluate the performance of three machine learning algorithms: support vector machine (SVM), random forest (RF) and gradient boosting classifier (GBC). The classifiers will be evaluated on their precision, recall, and F1-score. The three ML models will be evaluated on two datasets containing data produced by the CI-system at Spotify, one dataset of 4432 build logs is used for training and validation and 8927 build logs is used for testing, a total data size of 5.2 GB for both datasets. The goal is to see how well ML can discriminate between infrastructure and user related problems.

1.2 Outline

Chapter 1 will start with an introduction to the problem this thesis aims to solve and related work in the field of text classification. Chapter 2 will start with a background, explaining the field of ML, CI and text classification. This chapter then continues by explaining the methodologies for the techniques used in this thesis, from preprocessing to an overview of how the ML algorithms works and how to evaluate them. Chapter 3 will describe the workflow and the use of each component described in Chapter 2 starting with an architectural overview of each step in the process, followed by the construction of the dataset, the preprocessing, and the evaluation of the ML models. Chapter 4 will present the result for each of the ML models based on the evaluation metrics explained in Chapter 2. Chapter 5 will discuss the result produced in Chapter 4 as well as possible extensions of the work presented in this thesis.
2 Theory

This chapter will present the theory required to understand the method and result in this thesis. The chapter will give an overview of the three ML algorithms, the methods used for preprocessing data and how to evaluate the result from the models.

2.1 Background

Machine learning is the task of letting the computer solve a complex problem without explicit instructions, given for example only historical data to learn from. It can be applied to problems which are not easily solvable due to their complexity. For example, classifying an email as either spam or not spam, this requires the email subject and message as input and a label of spam or not spam as output. The task would be quite easy for a person who uses email regularly, but much more difficult for a programmer to solve by traditional programming. This is when ML can show its strength, given a large number of emails that are already classified as spam or not spam, an ML algorithm can create a sophisticated model to predict new emails as either spam or not spam based on the knowledge from prior categorizations [4].

2.1.1 Continuous Integration

Continuous integration (CI) makes the process of multiple developers working on the same source code easier. This is done by developers frequently commit code to the code base, and fail fast if a problem occurs, were the failure is detected by the CI-system. Using a CI-system and the rules of frequent commits, the developers focus less on fixing integration errors and more on implementing features. This leads to the developers working copy of the code is never far back in commits compared to the code in the version control system, due to the they frequently synchronize there code [1].

If the developers continuously commit their code to the version control system, then the CI-system will build a project multiple times per day. One of the CI-systems at Spotify is an internal product called Tingle, which takes care of orchestrating the process of retrieving the code and leaving feedback to the user and more. Tingle is using the popular open source building tool Jenkins to both test and build the code [3]. The CI-system together with Jenkins will produce large amounts of text that is stored in a build log. These build logs contain information about the source code and all the information from the beginning of the build until the end, with rich detailed information about each building step. These build logs are currently not in a format which makes it easy to read and understand for an untrained person, but is designed to provide more verbose information to the experts. These information-rich build logs are why the field of text classification is researched in this thesis, due to a large amount of information the CI-system produce is in a raw text format.


2.1.2 Text classification

Text classification is a part of machine learning where the aim is to categorize different types of text documents into a defined set of categories. This kind of classification ranges back to the early ‘60s but got its spotlight in the early ‘90s due to increasing interest in this field and the availability of more powerful computers [5]. This type of ML comes in focus now due to the large amount of information that is available online. With the help of text classification, companies now structure and handle their information in a new way that was not possible before [6].

2.2 Preprocessing

The task of preprocessing is a big part of text classification; it is the challenge to convert a large amount of text from the build logs into an ML friendly format. The job is to transform the text into numerical values, which then can be used as input to the models for both training and validation. This section will explain one technique which is used in this thesis to accomplish this transformation. This technique is called Term Frequency-Inverse Document Frequency also just known as TF-IDF, which is one of the most used weighting word technique in text classification [7].

2.2.1 Term frequency and inverse document frequency

The TF-IDF technique is a way to statistically calculate the importance of words in a document by normalizing the frequency of words using inverse document frequency. The term frequency part is done by calculating the frequency of each word in each of the documents. Only using TF will result in a vector where words like ‘The’ might have a high number (weight), due to it being a common word and words that do not appear often have a lower weight. The IDF part will take care of this issue by lowering the weights on words that frequently occur in the documents and increasing the weights on words that are less common [8]. The vector that TF-IDF produce can then be used as input to the ML algorithms.

The equation for TF-IDF is shown in Equation [2.1], where $tfidf(w)$ is the weight of the word $w$. The $tf(w)$ is the word frequency in the document, $N$ is the number of documents and $df(w)$ is the number of documents containing the specific word $w$ [9]

$$
tfidf(w) = tf(w) \times \log\left(\frac{N}{df(w)}\right)
$$

(2.1)

2.2.2 N-grams

N-grams can be used to take a sequence of N words as a feature instead of one word alone, for example a 4-gram would use *the car is brown* as one features instead of one feature for each word as in a 1-gram. Doing this may give a deeper insight of the words, then it would
be looking at each word individually. One example of a 1-gram and 2-gram extraction of the sentence "This is an infrastructure error" can be seen in Table 2.1 for 1-gram and Table 2.2 for 2-gram, where the ones are the number of times the n-gram appear in the sentence.

<table>
<thead>
<tr>
<th>Table 2.1 1-gram of the sentence &quot;This is an infrastructure error&quot;. The ones represent the number of times the n-gram occur in the sentence.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is an infrastructure error</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2.2 2-gram of the sentence &quot;This is an infrastructure error&quot;. The ones represent the number of times the n-gram occur in the sentence.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is is an an infrastructure infrastructure error</td>
</tr>
<tr>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

This thesis will evaluate 1-gram, 1-2-gram and 2-gram for each of the three ML algorithms. Where the 1-2-gram is a combination of both 1 and 2-gram. Given the amount of text in the build logs and the number of logs in the training dataset, using a n-gram over 2 would generate a very big TF-IDF vocabulary with a size over a million when doing 1-3-gram.

2.2.3 Stop words and minimum occurrence

One problem with using TF-IDF and n-grams are that the dimension of the feature space can become extremely large on big documents, reaching the order of $10^5$ or even $10^6$. To address this problem, we remove so called stop words, which is a set of common words provided by the scikit-learn framework [10]. These stop words are for example [a, an, the, of] and removing these will give the ML model more focus finding patterns on the relevant words [9]. Also, the TF-IDF processor used in this thesis will remove the words and n-grams that occur less than five times over all documents, due to the build logs contain a large amount of unique words/numbers/patterns that only occur in one specific build log, one example of this is the unique build id which is generated for each new build in the CI-system.

2.3 Machine learning models

Machine learning can be split up into three categories: supervised learning, unsupervised learning and reinforcement learning. Supervised learning is when each data point in the dataset comes with a given label; for example, a build log already labeled as an infrastructure or user error. Compared to unsupervised learning which is when the label is not known, and it is up for the ML model to try to find relationships between the data and try group them together, also called clustering. The final one, reinforcement learning is when each action is evaluated on a reward function, and the goal is to maximize this reward function.

The machine learning algorithms used in this study will focus on the field of supervised learning. Support vector machines is chosen in this study because of it being a popular algorithm often occur in the field of text classification [11, 12]. The second algorithm, Random forest, is also a popular algorithm in the field of machine learning, and works good
with high number of features [11, 12]. Random forest is a bagging model, which is a type of ensemble model described in Section 2.3.3. The last algorithm is chosen from the boosting group, which is the classifier Gradient Boosting [13]. After the preprocessing of a large number of build logs labeled as either infrastructure or user error, the supervised ML models can use the preprocessed data as input for training, testing and validation.

The following subsections will describe the general principle about the three supervised classification algorithms used in this thesis.

2.3.1 Support vector machine

Support vector machine (SVM) is a ML algorithm widely used in text classification. One of its strengths is that it works well on high dimensional data, which often is the case in text classification. The objective for the SVM is to find a hyperplane splitting the data, creating two distinct groups. It does this by trying to maximize the sum of distances between the data points and the hyperplane in a linear or high dimensional vector room. The data points close to the hyperplane are called support vectors. Later when used to predicting data, the new data gets categorized depending on which side of the hyperplane it lies in. Figure 2.1 shows an example of a hyperplane on a dataset which is linearly separable with two classes. The hyperplanes margin is shown with the support vectors backing up each margin line.

![Figure 2.1: Example of a two class dataset, seperated by a SVM hyperplane with marked out support vectors.](image-url)
2.3.2 Decision trees

Decision trees follow a tree-shaped structure containing decision nodes and branches leading to another decision node or leaf. Going down a tree checking each decision node, the categorization (label) will be what eventually lays in the ending leaf node. One simplified example of a decision tree can be seen in Figure 2.2, were the goal is to categorize the cause of a failed build log as either user or infrastructure. The example show two decision nodes, four branches, and three leafs. The classification of new data is done by going down the tree until a leaf node is reached, which will yield the label. For example in Figure 2.2, a build log would be classified as user failure if the text "User-error" is present and "Infrastructure-error" is not. One advantage with decision-trees is that the prediction of new data is extremely fast, follow the tree down to a leaf and you got your prediction [14].

![Figure 2.2: Example of a decision tree that categories if a build log failed due to user or infrastructure causes.](image)

2.3.3 Ensemble models

Ensemble model is a group of ML classifiers used together as one, creating a more powerful classifier by combining less complex classifiers [15, 16]. The ensemble model retrieves the predictions of all the classifiers, and the result of a majority vote decides the prediction of new data, one example of this can be seen in Figure 2.3. There are two different types of ensemble groups, there is the boosting, and the bootstrap aggregation also named bagging. **Bagging** generates sub-datasets from the large training set by randomly selecting B samples from the dataset (without removing them from the dataset) and then trains the decision tree with it [17]. **Boosting** is a sequential technique where each classifier is depended on the classifier before it. Where the classifier is trying to fit the data that the classifier before failed to classify correctly.
2.3.4 Random forest

The random forest algorithm uses the bagging approach explained in Section 2.3.3 and is constructed by decision trees explained in Section 2.3.2. Random forest adds an extra layer on top of bagging that performs random feature selection at each split. New data is then labeled by taking a majority vote of all the decision trees, and the final label is the one which is the most favored of all decision trees. Two advantages with the random forest algorithm, is its robustness to overfitting and that it is less sensitive to outliers [18].

2.3.5 Gradient boosting

Gradient boosting is an ensemble model that in this study uses decision trees described in Section 2.3.2. Gradient boosting is using the boosting method described in Section 2.3.3 instead of bagging that random forest uses. Gradient boosting is widely used and often gives a good accuracy when applied to different problems [19]. The algorithm comes with a cost when working with big data, like text classification in this case. One problem is that gradient boosting tries to find the best split to reach the highest information gain for each tree, which is a computationally expensive task and may take a long time to train [19, 20].

2.4 Evaluation

In this study, we will evaluate the performance of the machine learning algorithms on their precision, recall, and F1-score with help of a confusion matrix. One example of a confusion matrix can be seen in Table 2.3 followed by an explanation.
2.4. Evaluation

Assume that algorithm A is given a validation dataset X, and is tasked to predict the label Y for each data point in the dataset X as A(X) = Y. In this thesis X correspond to a tf-idf transformed build log and Y is the label user or infrastructure. Using this example and the confusion matrix shown in Table 2.3, the precision, recall, and F1-score can be calculated [21].

<table>
<thead>
<tr>
<th>User prediction</th>
<th>Infrastructure prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>User class</td>
<td>True positive (TP)</td>
</tr>
<tr>
<td>Infrastructure class</td>
<td>False negative (FN)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User prediction</th>
<th>Infrastructure prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>User class</td>
<td>True positive (FP)</td>
</tr>
<tr>
<td>Infrastructure class</td>
<td>True negative (TN)</td>
</tr>
</tbody>
</table>

Looking at Table 2.3, the prediction of a user label in the user class is called true positive (TP), prediction of infrastructure in the infrastructure class is called true negative (TN), which are the correct classification. Predictions of infrastructure in the user class are called false negative (FN) and prediction of the user in the infrastructure class is called false positive (FP), which correspond the wrongly classified ones [21].

With these variables described above, the precision, recall and F1-score can be calculated as showed in Equations 2.2, 2.3 and 2.4. Given Table 2.3, the metric **precision** measures the proportion of user predictions that are actually true. **Recall** measures the proportion of correct user prediction over both user and infrastructure predictions, when it was the user. **F1-Score** will measure the harmonic mean of precision and recall, resulting in a single value describing the performance of the model.

\[
\text{Precision} = \frac{TP}{TP + FP} \tag{2.2}
\]

\[
\text{Recall} = \frac{TP}{TP + FN} \tag{2.3}
\]

\[
2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.4}
\]

One important thing to take into account is that the equations for precision, recall and F1-score needs to be calculated for both user and infrastructure. If one would only apply the equations on the confusion matrix shown in Table 2.3 one would only get the precision, recall and F1-score for the user class. One example of calculating precision, recall and F1-score on the confusion matrix shown in Table 2.4 for both user and infrastructure can be seen in Table 2.5.
Table 2.4 Showing how the confusion matrix may look after a ML model predicted logs as either user or infrastructure.

<table>
<thead>
<tr>
<th></th>
<th>User prediction</th>
<th>Infrastructure prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>User class</td>
<td>90</td>
<td>5</td>
</tr>
<tr>
<td>Infrastructure class</td>
<td>10</td>
<td>85</td>
</tr>
</tbody>
</table>

Table 2.5 Result of calculating prediction, recall and F1-Score on Table 2.4 for both user and infrastructure.

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>0.90</td>
<td>0.95</td>
<td>0.92</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>0.94</td>
<td>0.89</td>
<td>0.92</td>
</tr>
<tr>
<td>Average</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
</tr>
</tbody>
</table>

2.4.1 Hyper-parameter optimization

In the field of machine learning, a hyper-parameter is the variable that will impact the learning process of the ML model. Changing this hyper-parameter will effect may change the learning rate and patterns that the model create during training, and the goal is to optimize the parameters for the specific problem one tries to solve. Depending on the model, the number of parameters can vary, were some of the parameters are a discrete value, and other continues values. There are different types of ways to find the optimal set of hyper-parameters, and the one used in this study is a technique called grid-search. Grid-search will take one or more hyper-parameters and a range of possible parameter values and test all combinations of it, and the combination that scores the best will be chosen as the final model. The score is based on the precision, recall and F1-score. So for example, an ML model with two binary hyper-parameters that is feed into the grid-search, will result in 4 different models when testing all combinations [22].
This chapter will give an overview of the work process for the experiments in this study. It will connect the components described in theory from preprocessing data, to train and evaluate the models.

This experiment follows a standard approach used in ML, where the first step is to prepare the data by cleaning and extracting features, then train and lastly evaluate the model, an overview of this process can be seen in Figure 3.2.

**Figure 3.1:** Overview of the work flow for these experiments.
3.1 Dataset

There are two different datasets used in this thesis; the first one is used for training and validation, the second one is used for testing. Usually in ML, the dataset is split randomly on a ratio of around 70% for training and 30% for validation, where one data point in the training set would not appear in the evaluation set. This is also the case with this dataset, with the modification of a 70/30 split on each group instead. A group, in this case, is a subgroup of either the user or infrastructure class. Both the user and the infrastructure have a large set of subgroups that all belong to the user or infrastructure, but they differ in there build log content. So to get a fair distribution in both the training and validation set, each of these subgroups is split on 70% for training and 30% for validation. One example of this split can be seen below in Section 3.1.1.

This study will also evaluate the ML algorithms performance on a test set, containing failed builds that occurred after the train-validation set was created. This test set is used to see how the models perform on data not seen during either training or validation, giving a more unbiased view of the performance. In this test set, the user failures are the majority class by far, as seen in Figure 3.1. One must have this in mind when inspecting the result.

![Figure 3.2: Showing the training, validation and testing split.](image)

The final dataset can be seen in Table 3.1. This dataset is not a clear 50/50 split on the total of logs for both user and infrastructure due to the group splitting technique used. The test set, which is created using builds that happened after the training and validation set was created, has the user class as the majority as expected. This is expected because the user errors occur more frequently than the infrastructure errors.

<table>
<thead>
<tr>
<th></th>
<th>User logs</th>
<th>Infrastructure logs</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training set</td>
<td>1236 (46%)</td>
<td>1468 (54%)</td>
<td>2704</td>
</tr>
<tr>
<td>Validation set</td>
<td>990 (57%)</td>
<td>738 (43%)</td>
<td>1728</td>
</tr>
<tr>
<td>Test set</td>
<td>8801 (98%)</td>
<td>126 (2%)</td>
<td>8927</td>
</tr>
<tr>
<td>Total</td>
<td>11027 (82%)</td>
<td>2332 (18%)</td>
<td>13359</td>
</tr>
</tbody>
</table>

3.1.1 Splitting on groups example

An example of the splitting of groups can be seen in Figure 3.3, where infrastructure has the three subgroups error A, B, and C. The A group, in this case, have 100 errors which are the majority of the errors, compared to B and C having 11 and 6. Doing a 70/30 split without taking the groups in account can result in that all the errors in the minority group get added to the training set. Which then results in that the model only evaluated on errors from group A, which may result in a good performance on evaluation but bad on new errors.
3.1. Dataset

![Diagram](image.png)

**Figure 3.3:** Made up example showing three groups of errors which are all classified as infrastructure errors.

3.1.2 Preprocessing

Before preprocessing starts, the build logs need to be stored locally and labeled as either user or infrastructure. The result is a table of all the build logs, label, and their metadata. The next step is to parse the big amount of text that these build logs contain. This parsing is done by only taking the text which is the output of the build process, meaning that the overhead text produced by the CI-system is ignored.

The next step is to split the table into one training and one validation set; this splits 70% for training and 30% for validation on the groups explained in Section 3.1.1. One may wonder why doing this split before the TF-IDF process. Because the TF-IDF process creates a vocabulary for the term frequency part which will not change after it have been created. Resulting in that after using TF-IDF on the training set, the vocabulary and weights will not change. So when the TF-IDF process transforms the text from the validation set, it reuses the already created vocabulary and weights from the training set on the validation set. Meaning that words that occur only on the validation set and not the training set will get discarded, due to they are not known when creating the vocabulary. If both the training and validation set were used to create the vocabulary and weights, information about the validation set would be leaked, something called information leakage.

The TF-IDF process results in three different vocabularies based on which kind of n-gram option is used during creation. The weight distribution of all features can be seen in Figure 3.4, were Figure 3.4a shows the weight distribution over 51874 features using n-gram 1, Figure 3.4b shows 413822 features using n-gram 1-2 and Figure 3.4c shows 361948 features using n-gram 2-2.
The TF-IDF process is also set to remove English stopwords described in Section 2.2.1, meaning it removes words that most likely do not bring any value. These build logs have many words that are unique to a specific document, for example, the building ID which should only occur once. These words do not bring any value and will only make the number of features large and might mislead TF-IDF to give them high weights, and by that mark them as important. One example of this can be seen in Figure 3.5, which shows a plot of the weights without removal of words on the left and removing them on the right. The left plot shows a total of 1265699 features where the majority of them get high weights, due to TF-IDF think they important due to they do not frequently appear in all documents. The right image show a representation of when the stop-words and words that do not occur often are removed, resulting in a total of 57536 features.

**Figure 3.4:** Shows the weight distribution of each feature after TF-IDF process on n-gram-1, n-gram-1-2 and n-gram-2-2.

**Figure 3.5:** Shows the weights of each feature after TF-IDF process. Image a) shows without removing words not occurring in 5 or more documents and left b) with removal.

### 3.2 Training models

The training of the machine learning models is done the same way for each algorithm. The algorithm for this procedure can be seen in Algorithm 1. After each iteration of hyper-parameter tuning, the best model for the specific dataset and ML algorithm is stored for later evaluation. The hyper-parameter tuning works by testing different combinations of parameters and evaluating the test score of each one. The model that yields the highest score is the one chosen as the final model. This process will generate one model for each n-gram, resulting in 3 models for each algorithm and a total of 9 models for each dataset.
3.3. Evaluating model

The nine ML models created in Section 3.2 will be evaluated the same way; this process can be seen in Algorithm 2. The ML models will predict the label on the test and validation set described in Section 3.1. From the predictions of the ML models and the dataset in that iteration, the precision, recall, F1-score and confusion matrix can be calculated. The results of this can be seen in the result Chapter 4. There are 9 trained ML models as described in Section 3.2 and two datasets that will be evaluated, resulting in a total of 18 evaluations.

Algorithm 1: Algorithm for the training and hyper-parameter tuning for the ML algorithms.

```
1 models = []
2 n_gram_datasets = [n_gram_1, n_gram_1_2, n_gram_2_2]
3 ml_algorithms = [svm, rfc, gbc]
4 for dataset in n_gram_datasets do
5    for algorithm in ml_algorithms do
6       best_model = hyper_parameter_tuning(dataset, algorithm)
7       models.append(best_model)
8    end
9 end
```

Algorithm 2: Algorithm for the evaluation of each model created in Section 3.2

```
1 models = best_models_from_training
2 datasets = [test_dataset, validation_dataset]
3 for ml_model in models do
4    for dataset in datasets do
5       predictions = ml_model.predict(dataset, ml_model)
6       precision_score = calculate_precision(dataset, predictions)
7       recall_score = calculate_recall(dataset, predictions)
8       f1_score = calculate_f1(dataset, predictions)
9       confusion_matrix = calculate_confusion_matrix(dataset, predictions)
10    end
11 end
```
4 Result

This chapter will present the result for each machine learning model on both the validation and testing dataset. The result is divided into two sections based on dataset and then in sections for each model. The models are evaluated using their precision, recall, and F1-score as described in Section 2.4.

The performance of the models will be evaluated using two different datasets, as described in Section 3.1. There will be three different models for each algorithm based on the n-grams described in Section 2.2.2. This will result in a total of 18 models with 9 for each dataset.

The precision, recall, and F1-score for each ML algorithm, on the three n-grams, are displayed in a bar chart side by side, followed by the corresponding confusion matrix as described in Section 2.4. The single word n-gram is abbreviated as n-gram-1, and the n-gram combining one and two words is n-gram-1-2, and lastly, the n-gram of only two words is n-gram-2-2.

This chapter will end with showing result for random forest and gradient boosting on 1634 build logs which are not labeled with regular expressions at Spotify, meaning the error cause is unknown.

4.1 Result from validation dataset

This section presents the result for support vector machine, random forest, and gradient boosting on the validation dataset described in Section 3.1. This dataset is created during the 70/30 split on each of the groups explained in Section 3.1.1.

4.1.1 Support-vector machine

Figure 4.1 shows that the three n-grams perform very well on the dataset. The n-gram-1 performs best with a score of 0.93 on precision, recall and F1-Score, with n-gram-1-2 next and worst n-gram-2-2. These numbers are supported by Figure 4.2, where Figure 4.2a represent the n-gram-1. Comparing Figure 4.2a, 4.2b and 4.2c clearly shows that n-gram-1 successfully classifies the correct label more often then n-gram-1-2 and n-gram-2-2.
Chapter 4. Result

Figure 4.1: Showing precision, recall and F1-score for SVM on validation dataset for n-gram-1, n-gram-1-2 and n-gram-2-2.

4.1.2 Random forest

The random forest models gives close to perfect score on all n-grams, as can be seen in Figure 4.3. The precision, recall and F1-score is near very close to 1 for the three n-grams, but the best model is the n-gram-1-2 as can be seen in Figure 4.3, giving a result of 0.99 on precision, recall and F1-score. This is also shown when comparing the confusion matrix for each n-gram in Figure 4.4. Figure 4.4 shows that n-gram-1-2 in Figure 4.4b is the the one that classifies the most correct labels.
4.1. Result from validation dataset

Figure 4.3: Showing precision, recall and F1-score for RFC on validation dataset for n-gram-1, n-gram-1-2 and n-gram-2-2.

Figure 4.4: Confusion matrix for RFC on validation dataset for n-gram-1 in Figure a, n-gram-1-2 in Figure b and n-gram-2-2 in Figure c.

4.1.3 Gradient boosting classifier

Gradient boosting shows outstanding result on each of the n-grams as can be seen in Figure 4.5 with a high 0.98 score on precision, recall and F1-score for all n-grams. Looking at Figure 4.6 for closer inspection, shows that n-gram-1-2 in Figure 4.6b gives the most correct classified labels.
Chapter 4. Result

Figure 4.5: Showing precision, recall and F1-score for GBC on validation dataset for n-gram-1, n-gram-1-2 and n-gram-2-2.

4.2 Result from test dataset

This section presents the result for support vector machine, random forest, and gradient boosting on the test dataset described in Section 3.1. This dataset is created from failed builds that occurred after the failed builds in the training and validation dataset. Meaning this dataset will give a more accurate view on how well the ML models perform in the real
4.2. Result from test dataset

world, as they are not split by groups like the validation dataset and there are for example new unseen projects.

4.2.1 Support-vector machine

The result shows that SVM do not perform well on the test dataset as can be seen in Figure 4.7 and Figure 4.8. The result shows that n-gram-1 performs the best with predicting the correct label, while n-gram-1-2 and n-gram-2-2 are dramatically worse if inspecting the Figure 4.8. This result suggests that SVM is not suitable to be used to predict the cause of build failures as either user or infrastructure.

![Figure 4.7: Showing precision, recall and F1-score for SVM on test dataset for n-gram-1, n-gram-1-2 and n-gram-2-2.](image)
Figure 4.8: Confusion matrix for SVM on test dataset for n-gram-1 in Figure a, n-gram-1-2 in Figure b and n-gram-2-2 in Figure c.

4.2.2 Random forest

The result show that RFC performs well on the test dataset as can be seen in Figure 4.9 and Figure 4.10. Only looking at Figure 4.9 show that n-gram-1 is the model that performs the best compared to the other two n-gram models. However looking at Figure 4.10 show another truth, were n-gram-1-2 predicts the correct infrastructure errors, but due to it wrongly classified a big amount of user errors, the score got lower.

Figure 4.9: Showing precision, recall and F1-score for RFC on test dataset for n-gram-1, n-gram-1-2 and n-gram-2-2.
4.2. Result from test dataset

4.2.3 Gradient boosting classifier

The result show that gradient boosting is performing well on the test dataset as can be seen in Figure 4.11 and Figure 4.12. When looking on the precision, recall and F1-score in Figure 4.11 the n-gram-1-2 model performs the best. The Figure 4.12 also show that the n-gram-1-2 model gives the most correct predicted labels of user and infrastructure.

**Figure 4.10:** Confusion matrix for RFC on test dataset for n-gram-1 in Figure a, n-gram-1-2 in Figure b and n-gram-2-2 in Figure c.

**Figure 4.11:** Showing precision, recall and F1-score for GBC on test dataset for n-gram-1, n-gram-1-2 and n-gram-2-2.
4.3 Classification on unknown build logs

The end goal for Spotify is to see if they can use machine learning to help label the build logs of the unknown type, meaning Spotify have not been able to categories the logs using regular expressions. The result of RFC and GBC of n-gram of one and two words to label 1634 number of unknown build logs can be seen in Figure 4.1. The true labels in the large set of build logs cannot be verified in simple way, so with a manual inspection of the infrastructure prediction using RFC show that 22 of the 26 errors were actual infrastructure errors and 11 of the GBC were also true infrastructure errors. The GBC model with n-gram-1 predicted a total of 59 infrastructure errors, resulted in 59 actual infrastructure errors when doing a manual inspection.

<table>
<thead>
<tr>
<th>ML algorithm</th>
<th>n-gram</th>
<th>User</th>
<th>Infrastructure</th>
<th>Confirmed infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFC</td>
<td>1-2</td>
<td>1608</td>
<td>26</td>
<td>22</td>
</tr>
<tr>
<td>GBC</td>
<td>1-2</td>
<td>1623</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>GBC</td>
<td>1</td>
<td>1575</td>
<td>59</td>
<td>59</td>
</tr>
</tbody>
</table>

Figure 4.12: Confusion matrix for GBC on test dataset for n-gram-1 in Figure a, n-gram-1-2 in Figure b and n-gram-2-2 in Figure c.
5 Discussion

This chapter will include the conclusions drawn from the results and future work.

5.1 Conclusions

This study shows that a Machine learning approach to categorize the cause of failed builds as either user or infrastructure is possible. However, this thesis concludes that the human factor is still needed to create regular expressions, but that machine learning can be used as a powerful tool to make predictions of unknown failures. The result from Chapter 4 shows that the most suitable machine learning algorithm is Gradient Boosting Classifier when comparing the precision, recall, and F1-score. The result also shows that using GBC with an n-gram combination of one and two words give the best result. After GBC, the random forest classified come close after with relatively good performance and support-vector machine with bad performance. The result shows that it was a big difference in performance between the validation and test dataset, meaning all the ML algorithms performed worse on the test dataset compared to the validation dataset. This may be because the machine learning models getting overfitted to the training data due to the group split explained in Section 3.1.1. The cause of the miss classifications in the test set may also be because of new projects being added, meaning new errors or even pipelines. This can lead to new words that do not exist while transforming the text in the TF-IDF process is lost, due to new words are not added after the transformation is done on the training dataset. The result also shows that using an n-gram combination of one word and two words give the overall best result; this can be because that n-gram 1-2 is the combination that gives the most information, which can be seen in the number of features each n-gram result in, described in section 3.1.2. The result can also be seen from another angle; it might bring more value to the CI-owners to use the model that performs best when categorizing infrastructure related errors then user ones. Picking the model with the best performance on predicting infrastructure errors might bring more value to the CI-owners due to it is the ones that are critical and can be fixed, while the user errors are only fixable by the users. During the process of this thesis, there have also come up build logs that have been wrongly classified by the original regular expression classifier; this most likely has a negative impact when training ML models, due to the ML models find false patterns. During a manual inspection of the 97 false positive predictions using RFC and n-gram, 1-2 showed in Figure 4.10, 23 of them were infrastructure error, showing that the ML model categorized the correct label correctly. The result from the predictions on the unknown build logs in Section 4.3 showed that ML can be used to help categories some of these unseen infrastructure errors. The results for the unknown logs also show that there most likely exist more infrastructure errors in the user predictions, due to the ML models find different amount of infrastructure errors. Taking these correct predictions in the unknown set and creating regular expressions for these unseen ones will most likely help the ML algorithms create a better model when retrained later on.
5.2 Future work

This study focused on the build text that was produced when building the code with Jenkins. The next step would be to only use the text from the step in the build that failed. This might increase the performance of the ML models and not misleading it with information generated by successful builds steps. Taking the last suggestion, using clustering algorithms on only the failed build steps may help the CI-Owners to create new regular expressions to help future machine learning models to predict better. When looking for the best ML model using hyper-parameter tuning, this step can in the future be done using cross-validation with K-splits and creating a new TF-IDF vocabulary with each split. This could result in a model which is performing better on real-time data and are not overfitted to training data. It is also possible to experiment with the hyper-parameters when using TF-IDF. TF-IDF was in this study used with an English stop-word vocabulary, which could be extended to remove words common in IT.


