CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Applying Machine Learning to Identify Maintenance Level for Software Releases

Master's thesis in Software Engineering

CHRISTOFFER STUART

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Applying Machine Learning to Identify Maintenance Level for Software Releases LaTeX

Christoffer Stuart

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Applying Machine Learning to Identify Maintenance Level for Software Releases
Christoffer Stuart

Supervisor: Miroslaw Staron, Department of Computer Science and Engineering
Examiner: Regina Hebig, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Applying Machine Learning to Identify Maintenance Level for Software Releases
Christoffer Stuart
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Maintenance is the single largest cost in software development. Therefore it is important to understand what causes maintenance, and if it can be predicted. Many studies have shown that certain ways of measuring the complexity of developed programs can create decent prediction models to determine the likelihood of maintenance due to failures in the software. Most have been prior to release and often requires specific, object-oriented, metrics of the software to set up the models. These metrics are not always available in the software development companies. This study determines that cumulative software failure levels after release can be determined using available data at a software development company and machine learning algorithms.

Keywords: Machine learning, supervised learning, unsupervised learning, defect prediction, cumulative failure prediction,

Acknowledgements

First of all, a big thanks to Miroslaw Staron for supervising and helping me through this process. The support and feedback has been appreciated, and without it this thesis would not have been possible. Thanks to Regina Hebig for being my examiner. Also, thanks to the telecom company that allowed me to do this project with them. Other people who have helped make this thesis a reality and to whom I am grateful: Frans Frejdestedt, Lars Ling, Wilhelm Meding, Anton Hemlin, Stellan Bondesson, and Kristin Wallenholm. Apologies for any I may have forgotten.

Last of all, but always first, thanks Elisabet and Christian Stuart.

Christoffer Stuart, Gothenburg, October 2019

Contents

1

Introduction

1.1 Background

60 percent of software costs stem from maintenance [18]. Therefore resource allocation for maintenance is an important question for any software company as too much allocation is costly, while insufficient allocation may result in poor customer satisfaction.

Maintenance is done for one of two reasons. It is either done to enhance existing products or to remove defects from the products. When it is done to remove defects from products there is an obvious correlation between the amount of maintenance needed and how fault-prone the product is. So, understanding how much maintenance is required at a certain point requires understanding how many failures are likely to occur in a product.

One common method of predicting maintenance levels is by using the opinions of experts within the organisation [43]. If experts are available this is the fastest and easiest way. However, this has two problems, first, the nature of the process: it is uncertain, possibly biased, and difficult to oversee. Secondly, the process itself is vulnerable, since the availability of one specific individual tasked with making the predictions may vary. For these reasons *data driven* approaches are more appealing. Data driven refers to basing decisions on data and empirical evidence rather than human judgements which are not clearly based on collected data. There is a variety of different options to make these data driven predictions, each with different strengths and weaknesses. One instance of data driven analysis is machine learning. This has gained increased usage in the industry overall and interest has increased to apply it to predicting maintenance levels as well [32]. For these models to be useful they need to achieve results matching or exceeding that of the experts while not subject to the above-mentioned drawbacks of using experts only.

A feature in a software program is a unit of functionality [3]. In the industry, developing software is often centred around developing features, and one team works on one or more features which are rarely cross-developed across more than one team. The errors in a program is often unevenly spread out across all parts, or features,

of the software [21]. Instead some features tend to be more likely to result in failure than others. The question is then if these software features can be identified before they start causing failures. If so, this enables increased scrutiny of these features before they are sent to customers. Also, it would be possible to spread out those features that are likely to fail across multiple releases so as to avoid sudden spikes in maintenance. Both of these approaches can thereby increase customer satisfaction and decrease maintenance costs.

1.2 Problem Statement

When developing large-scale software defects are inevitable [56]. Having a system to handle defects when they become failures is therefore important for any actor developing software. Poorly handled failures can result in high costs and loss of customer credibility. Whereas good failure handling in general, and proactive handling specifically, may decrease costs and increase credibility amongst customers. This can mean both finding and fixing *faults*, defined as some sort of error in the code that when triggered causes a failure of the programme, before they reach the customer. And to curtail the consequences of failures causing problems for customers. A software feature that is more likely to cause failures is also considered to be more fault prone.

The question is: How can consequences of software failures be reduced? For this two approaches are considered. First: to estimate how high the failure rate is likely to be within a release. And second: by increasing granularity to understand which of the developed software features in a future release are more fault prone.

Estimating the failure rate of a release aids organisational preparations for maintenance work. This means that the organisation can allocate resources to maintenance ahead of time if estimates show that a failure increase is imminent. Thereby the organisation will be able to remove defects quicker and lessen the impact of the failures for their customers.

Understanding how fault prone a certain software feature is enables an organisation to spread those features deemed likely to contain more faults over multiple releases so as to avoid sudden spikes in inflow of failures requiring maintenance. This could make maintenance allocation more predictable and efficient. Also, this can increase customer satisfaction as the company's products are perceived as less fault-prone; and an increased ability to deal with maintenance promptly. Such an approach requires a system that can predict short-term future failures, and group developed software features into categories ranked on their tendency to cause errors. Such a system should be objective, easy to understand, and correct (to the extent that is possible and feasible). This because it needs to be easy to compare results across the organisation, easy to implement, and useful, respectively.

1.3 Aim

There are two aims to this thesis. One has to do with failure-prediction, the other with defect-understanding.

The goal of the failure-prediction is to apply and compare five machine learning models that predict the number of reported failures in a release to find the best model. The models will be evaluated on accuracy in defect prediction. For the company this will help improve their process for determining current and future maintenance levels by making the process more transparent and consistent. Some of these models are *white-box*, their internal process can be understood, which can be used to understand what data they consider more important. This is considered beneficial since it can further help the company understand the underlying factors but not necessary.

The goal of the defect-understanding is to see how features can be grouped together using clustering algorithms. The intention is to then investigate if these groups vary in how much they contribute to defects. The end goal is to understand if there are certain characteristics shared among features that makes them more likely to cause failures. By doing this the company can improve their release strategy, by, for instance, spreading fault-prone features out over multiple releases. Thereby avoiding sudden spikes in maintenance levels. Further, this can support them in proactive decisions such as increased scrutiny of certain features prior to release.

1.4 Limitations

In this thesis the work done in a software development unit at a large telecom company is studied. They develop globally used software with high-demands on uptime. Only one product is considered. The data from this product comes from the reported failures, in the form of trouble reports, either generated from customers or internal testing. No distinction is made between the two. These trouble reports include data regarding when the failure was reported, how important the failure was to the system functionality, if it needed to be fixed, if and when it was fixed, and what release it was found in. Note, that this is not necessarily the same as what release the fault was introduced.

The data also includes information about the features developed. Both meta-data such as development time, and data of the commits to the repository containing the program. The central repository of the software program is called the main branch, this is cloned, or branched of, when a new release is created that the company's customer can then start to use after this branch-of has been tested and deemed ready for release. The commits are the uploads of the development teams to the branch. The commit structure is inconsistent. So some teams commit once per feature and other multiple times. This could mean that some data is missing from certain features if these have been uploaded with incomplete information.

2

Theoretical Background

Machine learning can be very broadly defined as "computational methods using experience to improve performance or to make accurate predictions." [37]. Where experience is the past information, usually in the form of data, that is available. All the considered solutions to the problem under investigation use different instances of machine learning. The two types used are *Supervised Learning*, and *Clustering*, which are explained below. Further, both types have different algorithms with different strengths and weaknesses, those algorithms which were considered as solutions are also expanded on in their respective sections. Finally *scalers*, which are used to format the data so that its usable for the algorithms are explained as well as *metrics*, used to rate and compare the results of the algorithms

2.1 Supervised Learning

In general, machine learning predicts some output based on input. Specifically for supervised learning, some of the input is labelled, meaning that it has a predefined output [23] [45]. Calling the input x, and the connected output y, we can then model the connection between the input and output as some function f, such that $y = f(x)$. A supervise learning algorithm then, is a function $\hat{f}(x, \theta)$ that approximates f:

$$\hat{f}(x, \theta) \sim f(x)$$

Where θ is some parameter(s) in the algorithm. During the *training phase*, which is specific to supervised learning, θ can be updated based on the accuracy of the approximating function. This updating is done to try and improve the models performance and is based on an update policy, and the *error* between proposed output $\hat{y}$, calculated by the algorithm, and the true output y. When the training phase is concluded the objective of the algorithm is then to make predictions without this feedback on its performance as it is now possible to use the algorithm on unlabelled data, meaning that the for an input x it is not necessarily the case that a connected output y is known.

The error, e, for one output-input pair which can be used as a performance measure for one update is:

$$e = ||\hat{y} - y|| \tag{2.1}$$

Where $||\cdot||$ signifies a norm, normally ℓ_1 or ℓ_2 [17].

There are two types of supervised learning problems considered in this work, classifier and regressor problems. A *classifier* is a type of problem aiming to predict *nominal* data, which belongs to certain group without internal order such as blue or grey. Or *ordinal* data, which is grouped data with some sort of order, such as small, medium, large. A *classifier* algorithm then is concerned with classification of some data set into classes [46].

A *regression* problem is that of finding a function which conveys the relationship between two or more variables. This can for instance be done when given a set of related points of $\mathbf{x}$ and $\mathbf{y}$. To estimate a function between the two sets is a regression problem. The estimated function is sometimes, referred to as the fitted line *Regressors* are algorithm which solve these problems of estimating functions [46].

2.1.1 Artificial Neural Networks

Real-life neural networks, like the brain, can be imitated by an artificial ones. An *Artificial Neural Network*, ANN, is a type of supervised learning algorithms

2.1.1.1 The Neuron

The first step in creating an ANN is to set up the basic unit, the neuron [34], the artificial neuron can at a time step t take N inputs and calculates one response which it then fires as its output.

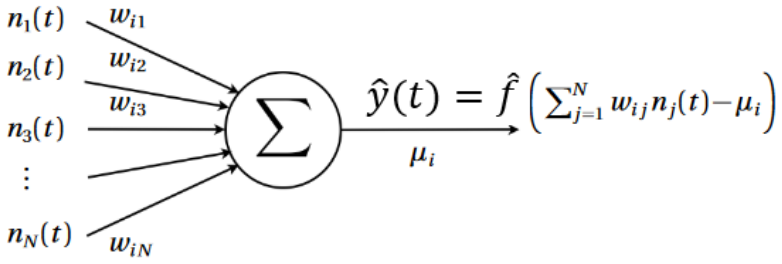


Figure 2.1: Example of neuron i in a larger network at timestep t, calculating an output $y(t)$ based on inputs n_j, weights w_{ij}, and threshold μ_i

In figure 2.1 can be seen one neuron that either acts on its own or can be part of a larger network of neurons. Actually, the single neuron can be used as a simple supervised learning algorithm. It takes some set of inputs, $\mathbf{n}$, of a predetermined size $N \times 1$ which it puts into a function $\hat{f}(x, w_{ij}, \mu_i)$ and generates an output, $\hat{y}$. What the function does is, sum up the inputs weighted by the set of weights w_{ij}, subtracts the threshold μ_i and finally applies some function to it. The final function

applied is called the *activation function* and can vary but one common option is the *Rectified Linear Unit*, ReLU [16]:

$$\hat{f}(z) = \text{ReLU}(z) = max(0, z) \tag{2.2}$$

other common options are the *Heaviside-* or *sigmoid*-function (equation 2.3), *linear*, and *tanh* [35].

$$\hat{f}(z) = S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{2.3}$$

As with any other supervised learning algorithm the network, or neuron in the simple case, can be improved through feedback on its performance. In a single neuron there are two sets of parameters which can be updated, the weights and the threshold function. In general the goal is to get as close to the labelled output as possible, thereby minimising the error as expressed in equation 2.1. Another way to express this is to minimise a cost function $J(\theta)$, based on the error by finding the optimum θ^*. Formally expressed as:

$$\theta^* = \arg\min_{\theta} J(\theta) = \arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} ||f(\theta; \boldsymbol{x}_i) - y_i|| \tag{2.4}$$

For some set of the training data of size m [19]. Finding this optimum is often done using *gradient decent*, which requires computation of the gradient:

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta || \left(f\left(\theta; \boldsymbol{x}_i\right) - y_i \right) || \tag{2.5}$$

The parameters can then be updated:

$$\theta_{new} = \theta_{old} - \eta \nabla_\theta J(\theta) \tag{2.6}$$

Where the small parameter $\eta > 0$, is called the *training rate*.

2.1.1.2 Multilayer Perceptron

For a simple classifier- or regressor-problem a single neuron could be sufficient. But the true power of the ANN is unleashed when combining these into large networks, called a *Multilayer Perceptron*, MLP, where the neurons, or *perceptrons*, are the basic units which are organised in layers. A layer feeds the next from input until the final output is generated. An example of this is shown in figure 2.2 where the MLP is *fully connected*, meaning that all the perceptrons in one layer connect to all perceptrons in the next layer, sending their calculated output via the connection. Note, in this *feed-forward* network, no perceptron sends its output to a perceptron in the same layer or the layer behind it. The example MLP takes five inputs, denoted by the black dots, then 3 perceptrons calculate their respective output and send that on to the final layer, consisting of two perceptrons who do similar calculations, thereby generating the final output in the form of a 2×1 vector.

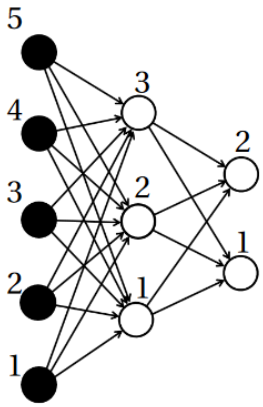


Figure 2.2: A Multilayer Perceptron taking fives inputs and generating a 2×1 output vector

Call the input $\boldsymbol{x}$, the generated output from the second layer $\boldsymbol{z} = \boldsymbol{f}_2(\theta; \boldsymbol{x})$, and the final input $\boldsymbol{y} = \boldsymbol{f}_3(\theta; \boldsymbol{z})$, where $\boldsymbol{f}_i$ is a set of functions containing each neurons specific function and parameters in layer i. The feed-forward nature of the network means that the output is dependent on all the steps between as in equation 2.7:

$$\boldsymbol{y} = \boldsymbol{f}_3(\theta; \boldsymbol{f_2(\theta; x)}) \tag{2.7}$$

In order to train the MLP in a similar sense to the training as described in the previous section we need to obtain the gradient for update for all the layer, despite only knowing the actual error for the final layer. The process of obtaining the corresponding correction for the other layers is called *back-propagation*. This is achieved using the chain rule:

$$\frac{dx}{dz} = \frac{dx}{dy}\frac{dy}{dz} \tag{2.8}$$

Applying it into the function describing the network gives:

$$\nabla_{\theta_2} J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta || \left(\boldsymbol{f}_3\left(\theta_3; \boldsymbol{x}_i\right) - y_i\right) || \boldsymbol{f}'_3\left(\theta_3; \boldsymbol{x}_i\right) \boldsymbol{f}'_2\left(\theta_2; \boldsymbol{x}_i\right) \tag{2.9}$$

When calculating the parameters for the hidden layer.

A certain subset of Neural Networks are able to feed some of the information back to itself in subsequent time steps. Meaning that as they calculate the output at a time step t this information can also be sent to the network as part of the input for a future time step $t + k$, $k > 0$. These types of networks are called *recurrent*, due to the data recurring in subsequent time-steps. This makes them excellent for,

amongst other, time series calculations as they also "remember" parts of previous input.

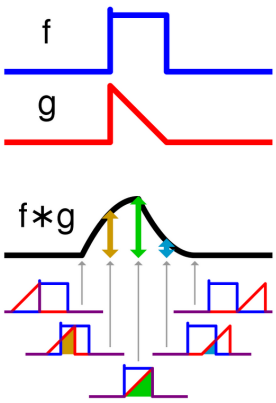
One issue when training neural networks is the *vanishing gradient*-problem. This occurs because the gradient calculated in the earlier layers tend to become smaller and smaller, thereby the update to the parameters becomes smaller, or may even stop. Recurrent Neural Networks are especially susceptible to this due to their adding of the previous data [5].

2.1.1.3 CNN

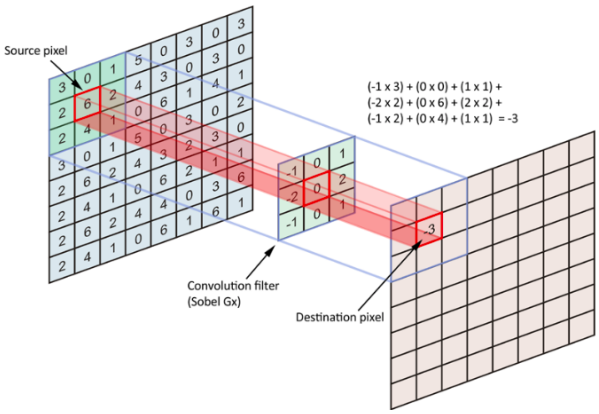
This type of Network is especially popular in image recognition [26]. And also has proven success on time series [8]. They are popular due to their usefulness on grid-type input, like that of an image. A CNN can detect specific data features, the presence of which are then summarised in another layer. The name comes from the convolution layer in which the convolutional operator is applied:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.10}$$

this has the effect of sliding the input function f over the function g which acts as a filter. This effect is imitated in a CNN as the input is "slid" through a filter in the form of the convolutional layer giving a similar effect.



(a) Example of the convolutional operator



(b) The convolutional operator as applied to a matrix with a 3×3 convolutional filter [12]

In figure 2.3a is an example of how the convolutional operator lets f be convoluted by g. It takes the form of g being slid over f and the resulting convoluted function is the area covered by both the function at a time t. Figure 2.3b, show how this is imitated by processing a source matrix by sliding the convolutional filter over the matrix, thereby creating the destination matrix.

2.1.1.4 LSTM

An LSTM, stands for Long Short-Term Memory (network), has the general structure of a neural networks in terms of the basic building block arranged in layers that feed the information from input to output. But the basic block is not that of a neuron as previously described. Instead the basic block is called a cell and contains an input gate, output gate, and forget gate [24]. This enables the cell to remember previous input. Making them very good in different time-series applications. The LSTM:s then are a variant of the Recurrent neural networks. But one of their benefits is that they get around the vanishing gradient problem.

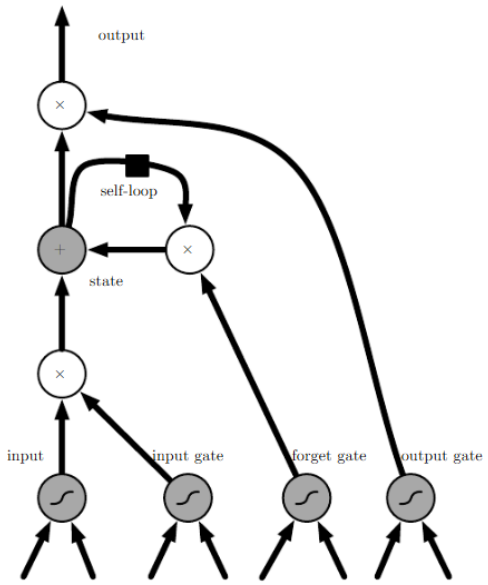


Figure 2.4: Displaying the inner workings of an LSTM-block, image taken from the Deep Learning Textbook by Goodfellow, et al. [19]

The different gates takes the states of the previous layer aswell as previous neuron states. This allows them to accumulate information regarding previous states. The input gate helps to regulate the current input, given to the LSTM-block, the forget gate regulates the internal feedback-loop of previous outputs (*self-loop*) and the output gate the final output. Usual functions for the gates is the sigmoid function.

2.1.2 LASSO

Stands for Least Absolute Shrinkage and Selection Operator. It attempts to fit the input data to the output data through estimation of a linear function who's coefficients it can update [46] [22]. If some input data is deemed irrelevant the coefficients are allowed to be zero. This function is achieved by minimising function 2.11 [62]:

$$\sum_{i=1}^{N}\left(y_i - \sum_j x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j| \tag{2.11}$$

Where **y** is the output, **x** the input β the weights subject to regularisation. The parameter γ is related to the shrinkage, a smaller γ implies less shrinkage of the less important variables.

2.1.3 Random Forest

Is part of a set of learning algorithms called *ensemble learning*, that combine the results of multiple algorithms to improve the performance by reducing the generalisation error [30]. RandomForests utilises a group of *Decision Trees* that differ by random changes introduced in the creation of the trees. Decision trees easy to interpret as they form a flow chart-esque algorithm for predicting or classifying the input patterns, makes data feature ranking easy and has shown some results with ordinal data [27], which is considered in the thesis. However, they are not very robust to data changes, finding the optimal tree is an NP-complete problem, meaning that there exists no algorithm for finding the exact solutions instead these have to be found with heuristics [9]. Also, they are prone to overfitting [30], see description in section 2.1.5. The last issue is somewhat mitigated by using the ensemble model.

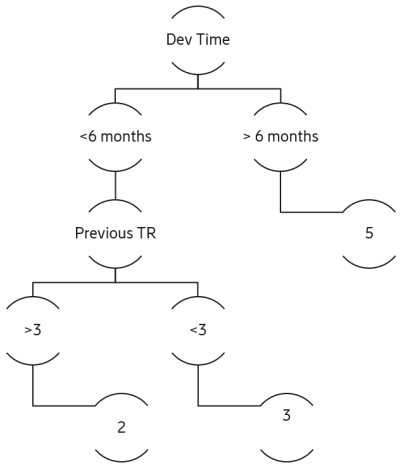


Figure 2.5: Example of what the logic flow could be in a decision tree on a classifier problem. Note that the full flow is not finalized

2.1.4 SVC

Support Vector Classifier works by constructing a multi-dimensional plane, *hyperplane*, that separates entries into different categories [37]. This is a very simple approach to any categorisation problem. SVC:s offer two main benefits. The first is that it selects the hyperplane with the largest margin separating the clusters. The second is that it allows for selection of *slack* variables, for which the constraint

penalty if certain data points are not on the "right" side of the hyperplane is lessened. Mainly, SVC:s are meant for binary problems, and require extending two work on multi-class problems [25]. These extensions are less anchored in theory and based on testing .

2.1.5 Challenges in Machine Learning

Noise in the data may obfuscate the underlying function. And when an algorithm starts to take the noise into account when calculating this causes *overfitting* [46]. Accounting for noise may increase the accuracy of the algorithm on the training set, while overall accuracy is actually deteriorating. The algorithm will then start to deviate from the underlying function. This can be discovered by separating a part of the data into a testing set, the accuracy of which is not used as feedback for the algorithm to improve. If the algorithm is improving on the training data but not on the test data this may be because it has started to account for the noise, it is overfitting.

Another issue is the amount of data required to solve problems using machine learning. While the amount of data required to train networks with multiple layers has decreased in recent years. A vast amount of data is still required to train algorithms for good performance [13]. Experiments in the field of medical research has created the "1-in-10", rule of thumb for regression problems. Which states that 10 events are needed for every 1 parameter that is to be explained. However, some research has shown that it may not be enough, depending on the specifics of the data used [11].

2.2 Clustering

Distinctly different from Supervised Learning, Clustering requires no output data. Instead it takes the input and by methods of comparison decides if what data points are similar to eachother [46]. Both types here are similar to classifiers mentioned in the section on supervised learning in that they group the supplied data into nominal groups. Of course the difference being that there is no labels available.

2.2.1 k-means

Is perhaps the most commonly used clustering algorithm. The scheme is as follows: The number of clusters sought, k, is selected by the user. Then, in the data set k points are selected randomly. These form the centroids of their respective clusters. Every point is then checked for the distance to each of the centroids. Point are assigned as members to the cluster with the closest centroid. With all points assigned to a cluster new centroids are obtained by calculating the average position of all points in the respective clusters. With new centroids the process is then repeated. The repetition then goes on until a convergence-criteria is met. Common choices for convergence-criteria is cluster difference between recent iterations or number of iterations [46].

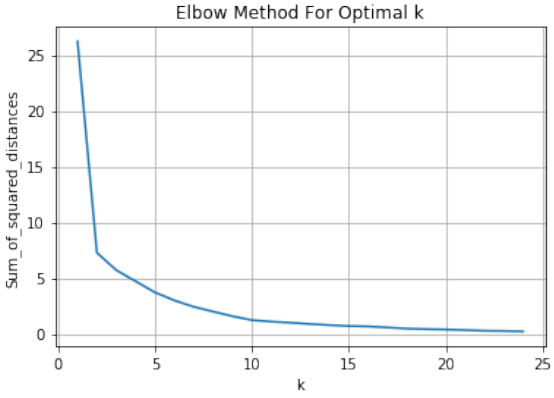


Figure 2.6: Plot of the elbow-method. Clear elbow at 2 clusters

One of the problems with k-means is due to the number of clusters being set in the beginning the algorithm will find that number of clusters, no matter what the actual number of clusters are. If the data is easy to visualise it may be easy to determine the number of clusters, thus mitigating the problem. But for high-dimensional(>3 dimensions) problems it is not easy to visualise and some other method is necessary. Often the *elbow method* [6] is used, this method is to run the algorithm for different numbers of clusters, usually starting from 1 cluster and increasing by a step of 1 for each run.

For every run the *inertia* is calculated by equation 2.12.

$$\sum_{i=0}^{N} \min_{\mu_j \in C} \left(\|x_i - \mu_j\|^2 \right) \tag{2.12}$$

Where $x_i \in X$, X is the set of datapoints of size N to be clustered, C is the partion of the set into k clusters, each cluster with a centriod of $\mu_j, j \in [1, 2, ..., k]$. This gives a measure of how close all points are to their respective centroid. The higher the number of clusters the less this measure will be. This measure is then plotted for each of the runs with the number of clusters on the other axis. This measure will always decrease for an increasing number of clusters. But, it decreases more rapidly for lower values and at some point, the elbow-point, this decrease becomes less rapid for each increasing cluster. This elbow point is considered the best number of clusters as the separation is considered good but not excessive as it would be by increasing the number of clusters further. Example is shown in figure 2.6

2.2.2 DBSCAN

Stands for *Density-based spatial clustering of applications with noise*. Though a long name this well describes what the algorithm does. It can take two parameters, one distance measure, ϵ, and one number, m, signifying the number of points required for a group of points to be considered a cluster [46]. The distance ϵ is the largest distance two points can have to each other and be considered part of the the same

group. If one of these points are within ϵ-distance of a third point this third point is also considered part of the group even if not all points are within ϵ-distance of each other. This enables the algorithm to find densities of more concentrated points and to find the appropriate number of clusters on its own. Further it can deem points not belonging to a clusters to be noise.

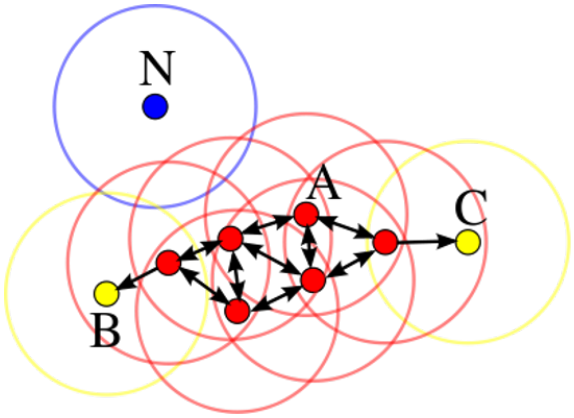


Figure 2.7: Figure showing how DBSCAN determines what points belong to a cluster, the circles denote the ϵ-distance

In Figure 2.7 the labelling in the DBSCAN algorithm is visualized. Note that points C and B are outside of eachothers ϵ-distance, yet are in the same group due to the connection to the points in-between. N is considered noise due to not being member of a group large enough to be considered a cluster.

The drawback of the DBSCAN-algorithm is that its parameters still need to be set and is therefore liable to create erroneous results if the parameters are incorrectly set.

2.2.3 Challenges in clustering

One major issue when clustering specifically but that can also create problems in other types of machine learning is the *curse of dimensionality* [46]. This occurs because as dimensions increase but the number of data points remain the same the dataset grows more sparse. With more space between the datapoints they become harder to cluster together. However, more dimensions means more information is contained in the dataset. Thus, keeping the number of dimensions low is necessary to avoid the curse of dimensionality. But, to retain information some balance is needed.

2.3 Performance Metrics

To rate the performance of the models different metrics are used. Note that these are also used by some of the supervised learning algorithms during the training phase. In such instances they are referred to as evaluation metrics. All metrics used are listed below

f1
f1 is used for classifiers. It calculates the *precision*, the number of instances of one category over the predictions of instances of that category. And the *recall*, the correctly labelled instances of one category over the total number of instances of that category [33].

MAE
Mean-average error is calculated by[46]:

$$\text{MAE} = \frac{\sum_{i=1}^{N} |y_i - \hat{y}_i|}{N} \tag{2.13}$$

MSE
Mean-squared error is calculated by[46]:

$$\frac{\sum_{i=1}^{N} (y_i - (\hat{y}_i))^2}{N} \tag{2.14}$$

The important difference between them is that MSE is much more susceptible to instances of larger distance. Meaning that as a metric its score will be significantly worse if the dataset contains a few datapoints which are of by a lot more than the others. Unlike MAE which is less affected by these large errors. But also thereby better at displaying the average error across the dataset, after which it is named [1].

R^2
Is a measure specific to regression problems, essentially it calculates how well the fitted line explaind the points. With a larger score signifying that it does a better job of predicting the points correctly. Of course this also depends on the nature of the points. With a points that are less scattered being easier to explain as some underlying function.

Silhouette score
This metric is specific to clustering problems that lack a ground truth. The score is obtained by creating a silhouette of each cluster based on the tightness of its internal points and separation from other clusters. The overall clustering is then rated on how well the points align to their specific cluster and lowered if overlap with other clusters exist [44].

$$s_i = \frac{b_i - a_i}{\max(b_i, a_i)} \tag{2.15}$$

Equation 2.15 describes how the silhouette score is calculated for one point s_i, where a_i is the mean distance to all points in the same cluster as s_i and b_i is the mean distance to the nearest cluster to s_i that s_i is not assigned to [48].

The silhouette score can takes values between [-1,1] where -1 indicates that there is a large overlap and 1 indicates that at the point s_i there is no overlap.

2.3.1 Scalers

Scalers are used to restrict the data into certain ranges. This can be useful for ANN:s with activation functions that can only act within certain ranges.

MinMax
The MinMax scaler [49] scales the input to within the range 0 to 1. The formula to scale $x \in \boldsymbol{x}$ to the scaled value x' is done by:

$$x' = \frac{x - \min(\boldsymbol{x})}{\max(\boldsymbol{x}) - \min(\boldsymbol{x})}$$

This preserves the internal distance between the datapoints, so is the least invasive. However, if the dataset contains distant outliers the main portion of the dataset is squeezed into a small part of the range[41].

Robust
Different to the MinMax this scaler has no actual upper or lower bounds. The operation is performed as follows:

$$x' = \frac{x - \text{median}(\boldsymbol{x})}{Q_3(\boldsymbol{x}) - Q_1(\boldsymbol{x})}$$

, where Q_i is the i:th quartile of the range of the set $\mathbf{x}$. The benefit of this scaler is that it is unaffected by outlier. Allowing the main portion of the dataset to be more spread out.

Quantile Transformer
This is also a robust transformation, meaning that the central values are spread out and the distance to the outliers reduced [50]. It can transform the values to both a normal distribution and a uniform distribution. There are some drawbacks using this, such that it does not restrict the data to within a certain range, and that it may transform the data in ways that distorts the information. Quantile transformation can be done to a uniform or normal target range [4].

3

Related Work

Defect prediction in industrial software projects is a heavily researched field, with many different methods available [43]. This is based around predicting the general inflow in a project or what parts are more likely to generate defects. The 80-20 rule regarding software defects says that 80% of software defects occur in 20% of the software's features [21]. Knowing that is only useful if you know what those software features are. Therefore, another related field is software feature clustering, results there have mostly been used in different re-use projects [57] [52].

3.1 Measure

A systematic review of fault prediction metrics by Radjenovic, et al. [40] shows that the choice of prediction metric that is used in fault prediction is important. There are three main types of metrics used:

- Object oriented metrics
- Non-object oriented metrics
- Process metrics

The first two prediction metrics measure the developed program itself, and are more based in traditional types of measurments: length, size, complexity and cohesion [59]. While the latter measures the process by which the program was developed. Object oriented metrics relate to object oriented programming and the structure within the programs built. Possible measures are for example: number of public measures, of inheritances, couplings, or methods per class [54].

Non-object oriented relate to metrics like size in terms of lines of code or complexity. Complexity measured for example by the MacCabe complexity measure, which measures independent paths through program [31].

Process measures relate to different change metrics, such as revisions, bugfixes, age, and others. These change metrics have been determined to give good results in fault

predictions [38].

The review [40] determined that object-oriented metrics tended to give the best performance and often correlated with non-object oriented metrics, and is backed up by others [55]. Although object oriented metrics capture more than non-object oriented ones such as size or complexity metrics there is a correlation between them. This is natural as larger structures, of any kind, tend to grow more complex, lines of active code naturally add to the complexity [59]. However, though complexity measures are intended to capture more of the nature of a program it has been proven to, at least in some cases be worse than simple methods such as lines of code [36].

3.2 Choosing Metrics

The importance of choosing a suitable metric was shown in a study where a significant correlation was found between inheritance metrics and the number of faults in certain parts of the software [10]. The study found that the classes that were in an inheritance structure were roughly three-times more defect-prone than those that were not. Further, classes that were lower in the inheritance structure tended to have higher likelihood of defects. Using this metric, and other object-oriented metrics such as: measuring events, and states, per class helped them construct useful prediction models.

Others have used the object oriented approach to show that they can be used to predict failure-proneness very early in a project [28]. The drawback is that it is limited to object-oriented programming. But due to the fact that object oriented metrics do correlate with size and complexity measures it seems likely that these are also useful.

Since size and complexity are correlated. It is maybe not surprising that lines of code (LOC) has been successfully used as a fault predictor [64]. This has been shown to improve when process measures where added also [39]. Raising the possibility that a combination may be the strongest option if both non-object oriented and process metrics are available.

3.3 Defect prediction

This thesis is concerned with cumulative failure-prediction, predicting the full amount of reported failures in a software release. And to investigate if certain types of software features are more fault-prone. Correlating roughly to these aims in the field of defect prediction are defect inflow prediction[58], and software defect prediction [43], respectively. The first is concerned with an overall level of defect inflow whereas the second is on a certain product with varying degree of granularity.

A study in forecasting defect backlog showed that achieving results that are usable

in an industry setting need to be easy to understand. The study showed that understanding the overall expected trend of the defect backlog, and that course reliability is most wanted from the stakeholder. Thereby more esoteric and complex models may be less useful as the resulting information becomes more complex, and maybe more accurate but less easy to use [60].

3.3.1 Defect Inflow Prediction

This has been pursued using primarily methods of multivariate linear regression [58], proved to have an accuracy of within 72% for a prediction of 3 weeks by drawing from characteristics of the work packages in development. Another study done by Fenton and Neil [15] create two very simple regressive approach based on either lines of code or complexity. But then conclude by recommending Bayesian Belief Nets for failure predictions. The regression analysis has been further developed in predictions in test settings, also including data regarding the program, size and some complexity as well as testing rigour [61].

Another study used Bayesian inference to predict the defect inflow in an ongoing project [42]. They achieved better results using this approach than many other approaches, such as multivariate linear regression, expert opinions, analogy based estimation, the only drawback is that the distribution needs to be estimated, which can be troublesome depending on the data. The study achieved an error of 10-20%, a good benchmark to aim for.

3.3.2 Software Defect Prediction

Is the prediction of the likelihood of defects within certain parts of the software. The granularity of these predictions depend on the methods used [43]. Common choices when using the machine learning approach is, amongst other, tree-based approaches, such as random forest, neural networks and support vector machines [29]. It was shown that the accuracy was quite good for the first two, but due to their complexity in parameter-tuning and training makes the pay-off uncertain.

One study was on release failure predictions in Eclipse post-release using object-oriented metrics and calculated using univariate binary regression and univariate multiple regression. This was done to both predict number of failures and to predict the different severities of those failures [53]. The study also showed that predictions become increasingly difficult after release. This may be due to decreased development and that the failures are more rare, largely due to having been put through a testing phase prior to release. Another shows that Neural Networks performed well in predicting failure prone parts of a program once a suitable metric had been chosen [28].

Finally a systematic review of machine learning techniques showed that they tend to outperform linear regression models. The review showed that the most successful models tended to be Random Forests [32]. Likely this is due to the algorithms robustness to noise and its tuning capabilities, as shown by Guo et al. [20]. Their

study primarily used MacCabe metrics but recommended testing other metrics as well.

3.4 Software Feature Clustering

Clustering in the software engineering domain has been used to cluster software features [52]. The findings reveal that it is more difficult when no labels are available. Further, automated clustering is often less effective than experts. Though this last one is hardly surprising. Shah, et al. [52] show that the level on which the clustering is done can vary the performance strongly

Usually software clustering is used when historical failures data exists, that can be used as ground-truth. But software clustering has also been attempted in predicting faults without access to underlying data [7] [47]. It proved to be possible using k-means, aided by a method to determine the clusters centers before initialising the k-means algorithm. Though the results where satisfying it should be noted that it is very difficult to measure the results in an unbiased manner as there was no underlying truth.

4

Research Design

4.1 Research Questions

The purpose of the thesis is to investigate the possibility to combine software clustering with fault-prediction. This created three main research questions, as outlined below, with related subquestions. These questions also roughly relate to the work structure into 3 iterations which are further explained later in this section.

RQ 1: How to predict the number of failures in a release using Machine Learning?

The algorithms that will be tested are determined beforehand to limit the number of tested algorithms. This research question is the central theme of the thesis and reoccurs in the last iteration but using more input data as that is gained during the work.

RQ 2: How can clusters of software features be created using machine learning to aid in failure prediction?

When the initial prediction algorithms have been understood the aim is to increase the dataset used by using aggregated data about the developed features to hopefully increase performance, described in **RQ 3**. This question is judged on silhouette score as well as distribution of clusters were clustering that would result in very sparse input come iteration three was considered bad because it may mean that it is not being accounted for at all, especially in the regression problems.

RQ 2.1: What data regarding the software features is best used for clustering?

This questions occurs due to the general problem of the curse of dimensionality. In general more data and variables mean more information but due to the curse this data needs to be trimmed and a balance kept. The evaluation is based on the same parameters as research question 2, silhouette score and distribution.

RQ 3: Are certain clusters more likely to contribute to faults in the software, and if so can they be used to increase performance of the prediction models?

This question combines the work done to answer the previous questions. By combining the prediction models of question 1 and the clustering of question 2 this question should also answer what factors in a certain software feature may cause it to be troublesome. Within the company this is now mostly done by intuition and guesswork regarding factors such as size and development time of a feature. Leading to the final sub-question

RQ 3.1: What data features are most important as predictors for the number of failures in a release?

By combining these questions the aim of the thesis should be achieved: to predict the occurrences of failure within the software and to determine what types of software features are more likely to cause these failures.

4.2 Research Methodology

The work-flow was divided into three iterations. Each iteration was concerned with one of the research questions, and their respective sub-questions, listed above. Question 1, using basic data, and question 2 could be resolved independently so this division of the work into iterations was useful and could be done without overlap. However, since research question 3 drew heavily on the insights gained in previous iterations and created new demands on the previous iterations. Mainly the second iteration was liable to changes in the third as that was the opportunity to test the second iteration results.

4.2.1 The Design Cycle

The process of each iteration follows the design cycle, proposed by Wieringa, R.J. [63] and described in Figure 4.1. The design cycle is an abbreviated version of the engineering cycle as Wieringa does not consider treatment implementation as part of the design process. Problem investigation outlines the process of understanding stakeholder interests as well as the nature of the problem. This prepares for treatment design, which refers to understanding the specific requirements, which ones of those helps to achieve the goal. This also includes mapping out what current treatments to the problems exist and create new treatments. The final step of the process is to attempt to validate if the treatments are applicable to the problem and context and can produce the sought effects. Usually the process requires some amount of repeating which is why it is shown as a circle. But in general it starts with Problem Investigation as described.

4.2.2 Data set

The entire data set was made up of information from 10 years back. Of this, the last four years were used, because the company switched release strategy at that time. Previously, they used to release new software features bi-annually, this was changed to monthly releases in mid-2015. It was clear that this caused a strong difference in

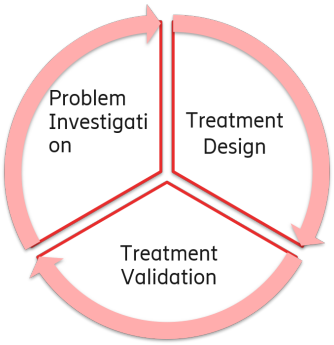


Figure 4.1: The Design Cycle

the cumulative failures in the releases before and after the switch and that it may end up confusing the models. The amount of failures per week was on average less than half after the switch to monthly. So, it is similar to training the models on two entirely different data sets with different underlying structure. A few others also had to be excluded due to lack of commit data. Thus, the subset of releases used was 40.

The data set used was made up of four subsets:

- Trouble reports
- Software feature data
- Commit data
- Miscellaneous information

Miscellaneous information was data such as lists of dates for releases and other information that could help connect the other data sets. For instance, this helped map commits that made up a software feature to a release by checking the date of the final commit and comparing to the next available release.

Trouble reports contained information about the detected failures, these could be detected either through internal testing or externally, by a customer. Every trouble report contained information regarding: when the failure occurred, in what release, and how severe it was. It also contained other information, but what has been mentioned is what was used. The number of generated trouble reports connected to the releases investigated was above 5,500, of varying severity. The majority of the failures were discovered prior to the actual release of the features by internal testing.

Software feature data contained high level information about the features, what date it was finished, how long development took. While all features had ID:s to separate them, these ID:s were not the same as the ID:s used to label the features

in the commit data subset, see below. The total number of software features in all investigated releases numbered just above 2,000.

Commit data contained information about every commit made to the main branch. What software feature it was part of, when it was done, what specific code changes were made, such as number of lines added, removed, or changed, and where in the system, those changes were made. Depending on the development teams' ways of working one commit could include multiple changes to multiple parts of the system, subsystems or be limited to one specific part. All commits were grouped together with other commits of the same ID. Most (>85%) of software features added or removed less than 1,000 lines of code. 19 subsystems were commonly occurring across the software features. Those subsystems that occurred less than 13 times each across the entire data set were grouped into one individual group, called *Rare Occurrences*.

One major issue was that there was no clear connection from individual commits or software features to an eventual failure report. But, commits could be aggregated into the individual software feature that they made up. What release a software features is part of was determined by the date that it was finished, and what the next available release was. A release is created a month before it is made available by taking a copy of the main branch. This copy is subjected to tests and fixes (as determined by the tests) for a month and then made available to customers. The method for attributing a features to a release was done by checking last upload date in that feature, verifying the next possible release, if that release (Release n)was less than a month away the feature would be attributed to the release planned after (Release $n + 1$).

The aim is to predict quality in terms of number of failures per release. So, it was decided in iteration 3 that the best way would be by counting the number of failures as attributed per release. This kept the time series nature of the data set, since the releases are ordered in time, Release n made available one month before Release $n + 1$. With the added benefit that when the actual failure was reported could be discounted. So, to the extent that it was possible (see section 5.1.3), a failure was attributed to the correct release by the algorithms.

4.2.3 Iteration 1

The aim of iteration 1 was to use basic data to tentatively answer research question 1. This was meant to help understand the problem, what data was available and what approach was likely to give good results. Here 4 of the 5 supervised learning algorithms were decided, with one allowed to be decided later for iteration 3. Specifically the basic data excluded more detailed data regarding the software features. In this iteration release as the time unit had not yet been decided upon so different time units where attempted.

4.2.4 Iteration 2

This iteration is aimed at answering questions 2 and 2.1. This included feature extraction and selection. More detailed questions that were part of iteration 2 included:

How to represent the software features?
When a feature is finished it is merged into the main program. This means that lines of code are added, changed or removed in different files in the repository of the program. Two main ways where considered. The first on a systems level, and the second on a subsystems level. For both of these the number of changes was calculated in two different ways, by the number of files changed or the numbers of lines of code. Making four different representations in total. Performance was then compared using the silhouette score.

What scalers to use?
The scalers that where used are listed in section 2.3.1. These were chosen to represent different options and varying degrees of invasiveness to the original datastructure. The scaling can affect the outcome of a clustering algorithm as the scalers affect the distance between the points and the clustering algorithms use this distance to perform their algorithm. Once again the silhouette score was used to determine the performance of the scalers.

Which algorithm to use
The DBSCAN and k-means algorithms were used. Since k-means is in such wide use omission would require strong reasoning based on the type of problem. DBSCAN was chosen due to the assumed noise in the data since it should be able to handle this well.

4.2.5 Iteration 3

The final iteration aimed to answer research questions 3 and 3.1. The goal is to improve the models set up in iteration 1 by expanding the data input, and to understand if there exists types of software features that are more likely to correlate with, and/or, cause the reported number of failures.

To improve the understanding of what clusters have greater effect on the inflow of faults trials are run with some of the clusters removed if this affects model performance. It may be that if clusters can be removed without affecting the models performance those clusters are less likely to cause failures.

For iteration 3 one of the main concerns was that using releases as time unit strongly limits the number of datapoints avaliable. To get around this noise was added to some of the models in an attempt to augment the data. Also, to increase performance in the ANN:s grid search was used to tune parameters, such as dropout, and neurons in the layers.

5

Results

5.1 Iteration 1

The first iteration was aimed at answering **RQ 1**. This was done using aggregated data relating to the recently released software features. Mean development time for the features that made up the release, the number of released features within the release and previous inflow of reported failures. The lag was chosen to $t - 4$, this was decided through testing where larger lag was tested. But the algorithms which can rank their input consistently put input from further back in time below the top 10.

5.1.1 Setup

The algorithms, and their parameters are presented below:

LSTM

Tables 5.1 and 5.2 shows the set up of the two LSTM networks. The parameter value signifies the number of neurons in all cases expect the dropout. Dropout signifies a layer which removes a parameter value sized portion of the inputs from the previous by setting them to zero. It chooses which at random. This helps to

LSTM-categorical		
Layer type	Parameter value	Activation function
LSTM	100	tanh
LSTM	500	tanh
Dropout	.7	-
LSTM	700	tanh
LSTM	100	tanh
Fully connected	70	Linear
tanh	1	softmax

Table 5.1: Setup of categorical LSTM-network

LSTM-regressive		
Layer type	Parameter value	Activation function
LSTM	100	tanh
LSTM	100	tanh
Dropout	.5	-
LSTM	50	tanh
Fully connected	50	RelU
Fully connected	70	Linear
tanh	1	softmax

Table 5.2: Setup of regressive LSTM-network

prevent overtraining.

The regressive network used MSE as internal evaluation metric, and a mini-batch size of 30 to train the network. The network was run for 2000 epochs, but interestingly showed no indication of overtraining. However, improvement stopped after 600 epochs.

The categorical network used categorical crossentropy as evaluation metric and mini-batch size of 50. It was initially run for a similar number of epochs but due to overtraining the subsequent tests were halted after 75 thereby decreasing overtraining. The results for categorical are calculated after 75 training epochs.

LASSO
The parameter in LASSO is γ, which was set to $\gamma = 1$ which gives the algorithm a propensity to set less important featuers to zero.

Random Forest
Random forest created 10 trees from which it combined the results to create its output. No max depth was set, the number of possible nodes in one walk through the tree, instead the tree was determined to be large enough either because all leaves are pure or the smallest number of samples for split is reached, the smallest number was two. Purity calculations was done with gini impurity for the classifier and MSE for regressor.

SVC
The Support Vector Classifier was setup using the default parameters of *sci-kit learn* [51]

All used the inflow of failure data but transformed to classes between 1 to 6 relating to how big the difference was from the previous months inflow. Clarified in Table 5.3

Preprocessing for the regressive algorithms was also done. The data, both input and labelled output, was transformed using a MinMax-scaler rescaling the values to the

Category	Difference in inflow, x
6	$x > 30$
5	$15 < x \geq 30$
4	$5 < x \geq 15$
3	$-5 < x \geq 5$
2	$-15 < x \geq -5$
1	$x < -15$

Table 5.3: The classification of failure inflow depending on the difference in inflow between the current and previous month

range $[0, 1]$.

5.1.2 Results

Score		Random Forest		LASSO	SVC	LSTM	
		Categorical	Regressive			Categorical	Regressive
MAE	Full set	0.29	**0.04**	0.06	0.57	2.11	0.03
	Test set	1.36	0.09	**0.06**	1.8	0.96	0.11
R^2	Full set	0.66	0.91	0.87	0.32	-0.93	0.92
	Test set	-0.41	0.35	0.72	0	-0.56	-0.13
MSE	Full set	0.98	<0.01	0.01	1.90	7.09	<.01
	Test set	4.63	0.01	0.01	6.06	1.78	.02
f1	Full set	**0.88**	-	-	0.79	**0.57**	-
	Test set	**0.42**	-	-	0.15	0.22	-

Table 5.4: Performance from iteration 1

Table 5.4, shows the performance of the described models when applied to the iteration 1 dataset. Note that the for the categorical LSTM model the f1 score was 0.57 for the entire set. Meaning that it after weighting of the precision across the different labels the model mislabelled data almost as much as it labelled correctly. This is worse than the other categorical algorithms. Best in the categorical algorithms was the random forest algorithm, performing better on the full set and almost twice as well on the test set than the second-best. However, the performance on the test set in absolute terms was below 0.5, meaning more mislabels than correct labels.

Interestingly, the Regressive LSTM was, unlike its categorical counterpart, on par with the other algorithms of the same type. . This was true both for MAE and MSE, and according to the R^2 score explains almost all the variance of the output. The best performing algorithm was random forest on the full test set and LASSO on the test set.

5.1.3 Adressing Research Question

RQ 1: Which machine learning algorithm is best suited to predict the number of failures in a release?

The best performing algorithm across the test set was the LASSO. For any model it is expected that the error is somewhat increased from the training to the test set. Using the LASSO model this was very small, so it seems to have estimated the underlying data relations very well. Random Forest and LSTM did better on the full dataset using both MAE and MSE metrics, but increased significantly when just measuring the error on the test sets. The categorical models performed poorly, f1-score < 0.5, on the test set.

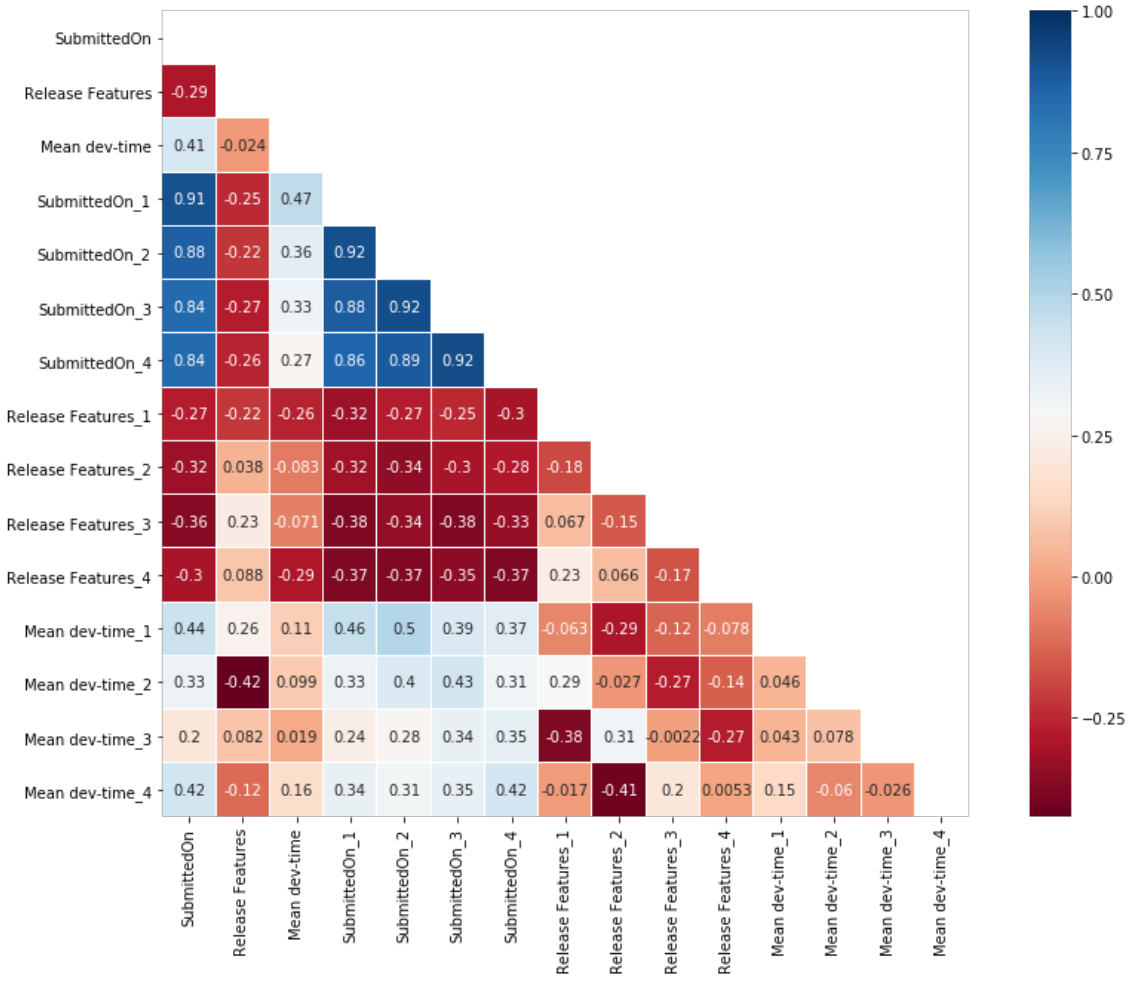


Figure 5.1: Matrix displaying covariance between TR inflow (SubmittedOn), number of released software features, and average development time of all features in a release. From time step t to t-4

It is possible that the poor results for the categorical models is that there is a strong auto-correlation for the overall trend of the inflow of trouble reports. Meaning that the inflow, while varying from month to month, in general is more likely to decrease if it has decreased previously, and vice versa if it has been increasing. Then the transformation of the inflow to categorical data removes some of this correlation. The strong correlation of the TR inflow with itself is shown in Figure 5.1 (see row SubmittedOn, columns $\text{SubmittedOn}_1 - 4$). This is further proven by Tables 5.5 and 5.6 that show the relative importance of the different data features for the Random

	Random Forest Classifier	
	Feature (at time)	Importance
1	Mean development time (t-2)	0.13
2	Mean development time (t-3)	0.12
3	Mean development time (t-1)	0.10
4	Number of releases(t)	0.08
5	Mean development time (t-4)	0.08

Table 5.5: Random Forest Classifier: Top 5 data features

	Random Forest Regressor	
	Feature (at time)	Importance
1	TR inflow(t-4)	0.36
2	TR inflow(t-1)	0.33
3	TR inflow(t-2)	0.10
4	Mean development time (t-1)	0.05
5	TR inflow(t-3)	0.05

Table 5.6: Random Forest Regressor: Top 5 data features

Forest Classifier and -Regressor, respectively. Based on the results the decision was made to drop the categorical LSTM model. The other classifiers were kept however. This was because they were less reliant on the TR data and therefore could be used to understand the relevance of the additional data in iteration 3. Further the decision to include a CNN model in the third iteration was made. Partially due to the faster training, and because they have been there are indications that they are more robust when transferring to other problems[2]. Which may hold relevance to the company when, for instance, applying the network to another product in the company.

Because trouble reports are tagged with the release in which they were discovered the decision was made to use the releases themselves as basis for the time series. One release may be in use for a long time before a customer decides to use a newer one this means that all the total number of failures discovered in a release may span a time period over a year, as show in Figure 5.2. An investigation into this behaviour shows that the average behaviour is that 63% of total failures are discovered prior to release. During the first four months after release 58% of the failures which are discovered while the release is in use are discovered. So the lag-time is very long. Undoubtedly, this means that customers update their software often will attribute the failures in a release to a later one. But using releases should minimise this compared to using weeks or months.

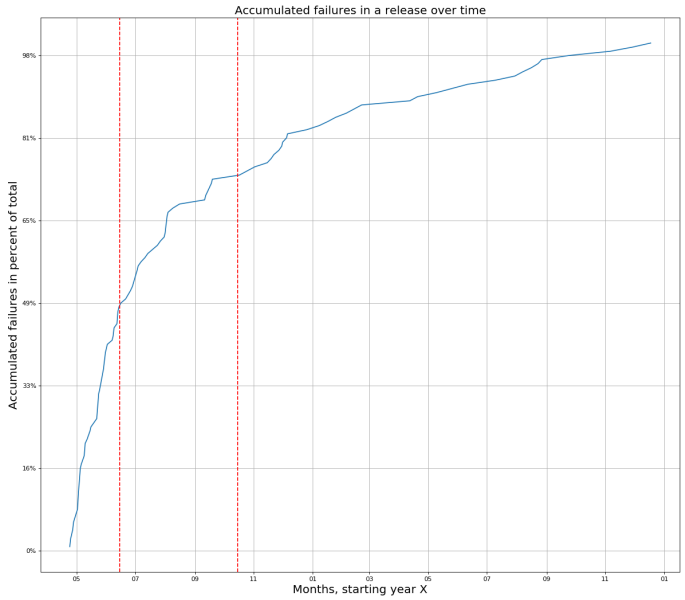


Figure 5.2: Graph showing cumulative inflow of trouble reports in percentage of total accumulated failures over time for a release made available in June of year X. First dashed vertical line is release date, the second dashed vertical line marks 4 months later

5.2 Iteration 2

5.2.1 Setup

Iteration 2 looked at data from the software features that made up each release. This data contained information regarding how many lines of code had been added, changed or removed in the systems as part of the creation of a new software feature. Also available was what systems or subsystems each code change took place in. What release the feature was part of was determined by looking at the what date the final change was made in the specific software feature. Thus clustering was done on the software features, as measured by the change done in the system attributed to the feature, the metrics are explained below.

There were two different versions of the clustering done in iteration 2. The two versions were different ways to represent the changes in the systems, attributed to the software features, to the clustering algorithms. The first was on a systems level. Each change that was made to a system was counted either by counting the number of files in that system that were affected or the number of lines of code. The second version was on a subsystems level were a similar calculation took place. Both versions were then pre-processed using the maxmin-, robust-, quantile uniform-, and quantile normal-scalers after which the clustering algorithms were applied.

Iteration 2 used k-means and DBCSAN as clustering algorithms. The number of clusters for k-means was chosen based on an assumption of the least number of categories needed for the third iteration, this assumption was 10 clusters. Also the

elbow method was used to determine the least number of clusters. This showed that 2 clusters would yield the best results to number of cluster ratio. Since both were considered minimum the higher number was used.

DBSCANs parameters where chosen by grid-search in the range $\epsilon = [0.1, 1.5]$, and $m = [5, 20]$. The best options were then selected by balancing the silhouette score with smaller number of clusters, < 30 so as to avoid feeding to much data to iteration 3.

The clusters were highly varying in size. This would have caused the time series formatted input data for iteration 3 to become very sparse. While there are more robust methods to use for regression problems with sparse data [14] it was deemed likely that this could be problematic for the models and may be unrepresentative for the reality of the situation. This was because while some software features may certainly be outliers it seemed unlikely that the main part of all features developed were not different. So for that reason the clustering methods performance was measured on silhouette score and cluster distribution.

5.2.2 Results and Addressing Research Questions

	System			
	Number of files		Number of lines of code	
K-means	Silhouette Score	Largest cluster	Silhouette Score	Largest cluster
MinMax	0.929	98,2%	0.985	99,3
Robust v1	0.424	66,9%	0.854	96,1
Robust v2	0.399	60,1%	0.906	96,6
Quantile Uniform	0.537	22,8%	0.481	25,3
Quantile Normal	0.522	22,6%	0.486	25,3
DBSCAN		Clusters		Clusters
MinMax	0.993	2	0.995	2
Robust v1	0.741	25	-0.205	2
Robust v2	0.741	25	-0.205	2
Quantile Uniform	0.771	28	0.677	28
Quantile Normal	0.776	28	0.724	29

Table 5.7: Results from clustering software features on system level

RQ 2: How can clusters of software features be created using machine learning to aid in failure prediction?

K-means and DBSCAN were tried and compared on the same datasets. First the software features were represented on a systems levels. One feature could affect more than one system and two different measures for the extent to which they affected all system was used. . The results on a systems level are in table 5.7 and results from subsystem level in 5.8.

The findings seemed to imply that there is quite a lot of noise in the dataset,

	Subsystem			
	Number of files		Number of lines of code	
K-means	Silhouette Score	Largest cluster	Silhouette Score	Clusters
MinMax	0.89	97,0%	0.95	10
Robust v1	0.13	79,2%	0.87	10
Robust v2	0.62	88,6%	0.87	10
Quantile Uniform	0.27	34,7%	0.26	10
Quantile Normal	0.25	43,3%	0.25	10
DBSCAN		Clusters		
MinMax	0.98	2	0.99	2
Robust v1	0.47	47	-0.55	16
Robust v2	0.47	47	-0.55	16
Quantile Uniform	0.48	2	0.49	2
Quantile Normal	0.36	20	0.34	20

Table 5.8: Results from clustering software features on subsystem level

which can make fault-proneness estimations more difficult [65]. DBSCAN has the ability to detect noise and outliers and that made up between 5-15% of the data on systems level depending on the pre-processor. For this reason it was decided that the clusters computed by the DBSCAN algorithm may be more appropriate than the ones generated by k-means.

RQ 2.1: What data regarding the software features is best used for clustering?

Due to the clustering performance being judged on the dual criteria of silhouette score and cluster distribution it was decided that the feature data detailing numbers of files changed on a systems levels was the best for clustering on.

5.3 Iteration 3

5.3.1 Setup

The data used in iteration 3 was an amalgamation of the data used in iteration 1 and that created in iteration 2. For the neural networks the layout was determined through a grid search, the result of which are shown in Tables 5.9 and 5.10, for the LSTM and CNN, respectively. The other algorithms had the same setups as in iteration 1. The data generated in iteration 2 was used and the output was increased to also consider how important a failure was considered. Ranging from 1, extremely important, 2, less important but still important enough to stop distribution of whatever feature caused it. 3 and 4 where important enough to fix whereas 5 was considered almost insignificant.

LSTM		
Layer type	Parameter value	Activation function
Gaussian Noise	0.05	-
LSTM	200	tanh
Dropout	.3	-
LSTM	150	tanh
LSTM	200	tanh
Fully-connected	70	Linear
Fully-connected	6	Sigmoid

Table 5.9: Settings of the LSTM-network used

CNN		
Layer type	Parameter value	Activation function
Gaussian Noise	0.05	-
Conv1D	64, 2	tanh
RelU	.3	-
Dropout	.3	-
MaxPooling1D	200	tanh
Fully connected	70	Linear
tanh	6	Sigmoid

Table 5.10: Settings of the CNN-network used

5.3.2 Results and Addressing Research Questions

RQ 3: Are certain clusters more likely to contribute to failures in the software, and if so can they be used to increase performance of the prediction models?

Feature rank		1	2	3	4	5	6	7	8
Correlation	Data Feature	**-1**	-1	**4**	4	**0**	3	9	2
	time step	-1	-2	-1	-1	-1	-1	-2	-1
	Importance	-0,69	-0,65	-0,59	-0,56	-0,55	-0,52	-0,52	-0,51
LASSO	Data Feature	20	7	Med	Long	20	17	22	13
	time step	-2	0	-2	-1	0	-2	0	-1
	Importance	0,25	0,24	0,22	0,21	0,18	0,13	0,13	0,12
Random Forest	Data Feature	**4**	**0**	**-1**	12	4	12	-1	2
	time step	-1	-1	-1	-1	0	-2	-2	-1
	Importance	0,13	0.11	0.11	0,07	0,07	0,06	0,06	0,05

Table 5.11: Data feature relevance for inflow of failures

Feature rank		1	2	3	4	5	6	7	8
Correlation	Data Feature	**-1**	-1	9	**4**	3	3	4	9
	time step	-1	-2	-2	-2	-1	-2	-1	-1
	Importance	-0,60	-0,55	-0,48	-0,45	-0,43	-0,41	-0,40	-0,39
LASSO	Data Feature	Long	Med	26	20	12	7	17	15
	time step	0	-2	0	0	-1	0	-2	-2
	Importance	1,4	1,0	0,81	0,79	0,74	0,58	0,48	0,45
Random Forest	Data Feature	**-1**	-1	9	12	Med	Med	Short	26
	time step	-2	-1	-2	-1	-1	0	-1	0
	Importance	0,18	0,099	0.083	0,072	0,057	0,041	0,037	0,036

Table 5.12: Data feature relevance for importance class 2

Feature rank		1	2	3	4	5	6	7	8
Correlation	Data Feature	**-1**	-1	**4**	4	**0**	0	3	9
	time step	-2	-1	-1	-2	-1	-2	-1	-2
	Importance	-0,62	-0,61	-0,58	-0,56	-0,53	-0,52	-0,51	-0,51
LASSO	Data Feature	12	Long	17	20	26	14	24	Med
	time step	0	-1	-1	-2	-2	-2	-2	0
	Importance	0,63	0,62	0,53	0,47	0,15	0,14	0,14	0,14
Random Forest	Feature number	**4**	**-1**	12	**0**	0	5	0	17
	time step	-1	-1	-1	-1	0	-2	-2	-1
	Importance	0,23	0.070	0.069	0,063	0,059	0,050	0,031	0,028

Table 5.13: Data feature relevance for importance class 4

Figures 5.11, 5.12, and 5.13 show what data features the algorithms use to predict the inflow of failures, and the inflow of failures of importance class 2 and 4, respectively. To answer the first part of research question 3, if certain clusters are more likely to contribute to failures in the software? The tables show the most important data features for total trouble report inflow and inflow of trouble reports of importance class 2 and 4. Some clusters of software features: -1 (noise), 0, and 4 are shared between Random Forest and correlation, but noticeably, for importance class 2 neither 0 nor 4 seems significant to Random Forest. LASSO though, utilizes other data features for its calculations. Noisey data has a randomness component too it, and often modelled by a normal distribution. In the case of DBSCAN though, the noisey data is separated from the other datapoints because they do not clearly belong to a cluster. So something separates them, it is not unlikely that software features that are significantly different from other previous developed software features should be more complex. The other clusters that reoccur across both Random Forest and Correlation may have some inherent properties making them more failure-prone.

Score		Random Forest		Lasso	SVC	LSTM	CNN
		Categorical	Regressive				
MAE	Full set	0.21	**0.06**	0.04	0.18	**0.05**	0.05
	Test set	1.0	**0.11**	**0.22**	0.809	**0.11**	0.16
R^2	Full set	0.76	0.86	0.68	0.82	0.86	0.80
	Test set	-0.01	0.31	-1.17	0.0	-4.46	-2.91
MSE	Full set	0.39	0.01	0.018	0.27	0.01	0.01
	Test set	1.89	0.02	0.091	1.33	0.03	0.04
f1	Full set	**0.86**	-	-	0.86	-	-
	Test set	**0.18**	-	-	0.17	-	-

Table 5.14: Performance iteration 3

The second part of research question 3, if knowledge of certain clusters that are more likely to cause failures can improve the models? To answer that Table 5.14 shows that no noticeable improvement occurred between the models in iteration 1 and 3. The LSTM network improves slightly and thus just becomes the best algorithm on the full set and equals random forest on the test set. But more importantly some actually got worse, noticeably the f1-score of the categorical Random Forest model was similar for the full set but on the test set had deteriorated significantly, from 0.42 to 0.18. Also, for the regressive problems LASSO performed worse than in iteration 1.

Regarding the first part of the research question, the three different ways to calculate feature relevance only coincide slightly. So while certain clusters do seem to be more likely to cause failures they do not seem to aid the models.

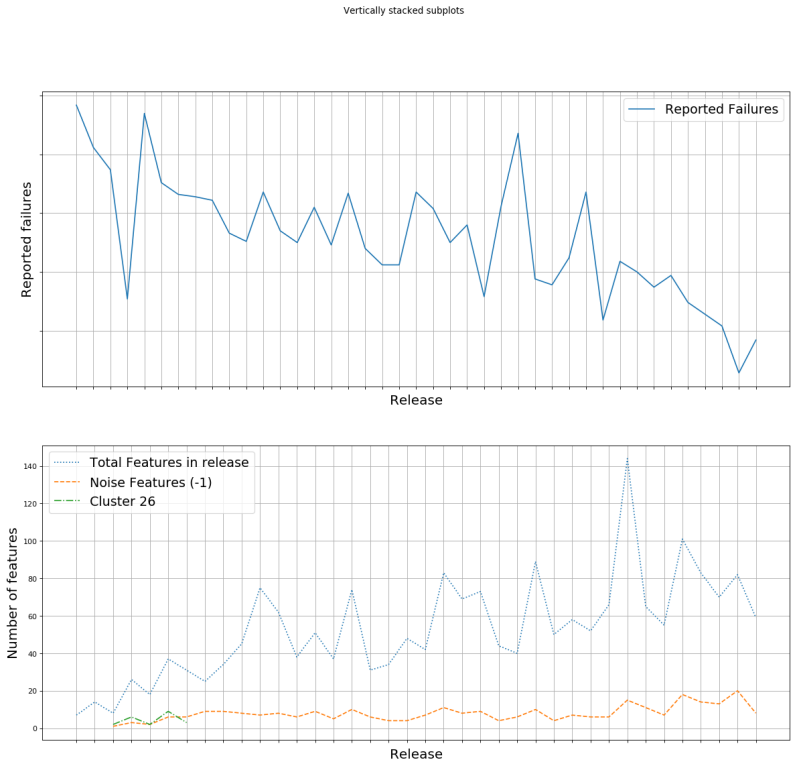


Figure 5.3: Top graph shows of accumulated failures per release (actual numbers removed due to company privacy concerns), bottom graph shows total software features delivered, and software features of type -1 and 26 per release included in the study.

Figure 5.3 shows the number of failures per release in the top graph. Due to company privacy concerns the y-axis has no scale. But it is still evident that the number of trouble reports per release has a downward trend, while developed software features has an opposing trend. So the correlation is negative, thus it is difficult to infer using a linear model what software features mostly contribute to increase in failures. Instead using such a model it would seem that more software features should mean less trouble reports. Noise is by DBSCAN labelled as cluster -1, this has a high negative correlation and is used in Random Forest as one of the more important data features. So it can be made to use predictions but the appearance of noisy software features in a release may not be interpreted as increased likelihood of failures.

6

Discussion

As is evident in the results there is a clear auto-correlation regarding the number of failures produced in the software. Meaning that while some releases result in more failures than others the overall trend is clearly decreasing.

One major issue for the models is that this auto-correlation makes it unlikely that the models will be able to detect sudden spikes due to their reliance on the historical development of features. What is interesting is that this could imply a strong causation of company policies and ways of working. But in order to clarify if that is indeed the case a wholly different study would also be necessary to map out what changes has taken place within the company.

Another issue for the predictions and especially to attempt to use the information gained from the algorithms which are able to rank their input is that the overall trend of feature releases is that they are increasing while the number of failures decrease. This means that the algorithms may conclude that more features mean less failures. This of course is not necessarily true. It could be, say that the features are decreasing in size giving the development teams better opportunities to scrutinise what they are producing but is not necessarily the case. And indeed, the fact that the company is expanding would instead suggest that their increase in features just has to do with more capabilities.

6.1 Data

One big issue remained after the data set had been trimmed for non-suitable data. This was that even using the miscellaneous data the different sets were only connected by implicit factors. Data was not available that made the connection between a feature, or release, and a trouble report regarding a failure explicit. This makes the solution somewhat unreliable as it is difficult to fully calculate the portion of the data that may be mislabeled. Such as a feature that was delayed for one reason or another. This added with the problem of lagging failures as presented in section 5.1.3 creates an uncertainty of how exact the data is. However, discussions with experts in the company place this as a rather small portion of the total set ($< 10\%$) and is not expected to change the overall nature of the data.

In iteration 2 some of the subsystems that were changed while creating new software features were re-labelled from their original name to rare occurrences. This grouped affected subsystems into one to avoid the curse of dimensionality. This hinders diagnosing specific subsystem under that label as particularly fault prone compared to others under the same label. But if that had not been done it is likely that it would have decreased the overall performance of the algorithms, due to the curse of dimensionality. So, this was done to allow for, and improve, general conclusions.

6.2 Metrics

From the related works could be concluded that object oriented metrics tend to be better. However this type of data was not available but for future work it may be necessary to improve the data gathering and feature engineering by accessing and extracting other data. This could include both object oriented, process and other complexity measures.

6.2.1 Ground truth

The two parts of the thesis differ significantly in their ability to be measured for their accuracy of their results. The prediction algorithms are possible to test on data with ground-truth, meaning old data. It turns out that they seem reliable and could therefore be interesting to use. In fact, even for the latest releases where they predict significantly higher number of failures than what has been reported this seems likely to be true. This is due to the fact the cumulative inflow continues, for most releases, for over two years.

For the clustering algorithm no ground truth exists. It could be calculated using different metrics but not without significant man hours put into categorising the different developed features in novel ways that could be meaningful to the clustering algorithms. This means that it is impossible to objectively determine their performance. And therefore no conclusion can be made if the results are accurate or not.

7

Conclusions

The thesis investigated cumulative failure prediction capabilities of certain machine learning algorithms. It also aimed to understand how software features can be clustered using available data. This clustering was used to see if certain clusters were more fault-prone. Further, the clustering was used to extended the cumulative failure prediction. These three steps formed the three iterations, initial cumulative failure prediction, clustering, and extended cumulative failure prediction. Of further interest in the cumulative failure prediction was what data features most contributed to the result. To solve this some of the used algorithms were white-box, allowing insight into how the data was weighted to calculate the results.

For the cumulative failure prediction five different supervised machine learning algorithms were used. The problem was set up in two different ways: as a classifier problem, where the number of failure was compared to previous number of failures and then classified according to size of increase or decrease. And as a regression problem, where the aim was to predict the inflow as is, although scaled to suit the algorithms. Although the regressors and classifiers were applied to different data sets and so should not be compared directly the regressors were accurate with regards to both the full set and test set (error $< 10\%$). The classifiers, specifically random forest which had the best performance of the classifiers, performed decently on the full set (f1-score of 88%) but classified less than half of the test set data correctly.

For iteration 1 the best performing algorithms were random forest and LASSO, on the full set and test set respectively. In iteration 3 LASSO performance on the test set deteriorated and in fact the best performance was by the LSTM-network, which underwent grid search to tune its parameters.

Clustering of the software features, which was done in iteration 2, is not possible to rate with regards to some absolute metric, due to the lack of a ground truth. It was clear that the data contained an amount of noise, so DBSCAN was deemed the better choice due to its ability to handle noise and put outliers in a category of its own. To achieve both good clustering results, rated using silhouette score, a reasonable distribution of number, and size of the clusters data regarding numbers of files changed on a systems level was used.

In iteration 3 the additional data created by iteration 2 was applied to the original problem in iteration 1 and extended to also investigate if certain clusters of software features were more likely to contribute to the likelihood of failures. Random forest, and correlation between number of features in specific cluster and the cumulative failure per release gave some similarities. But no clusters they indicated were the same as the clusters indicated by LASSO. These results were further obfuscated by the fact that the overall trend of failures is an decreasing number of failures per release, while the overall number of features is increasing. Further, the total number of failures attributed to one release is not determined until a year after that release has been made available to all customers. So, all features can be attributed as part of a cluster. And some of those clusters possibly contribute to increased likelihood of failures. But what those clusters are depend on the algorithm. The overall trends indicate that it is not only the complexity of a certain feature that determines likelihood of failure but also other organisational decisions that affect the work performed. As is indicated by the fact that despite more features being created they result in an overall net-decrease of failures.

Bibliography

[1] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. "On the Surprising Behavior of Distance Metrics in High Dimensional Space". In: *Database Theory — ICDT 2001*. Ed. by Jan Van den Bussche and Victor Vianu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 420–434. ISBN: 978-3-540-44503-6.

[2] Jonas Alexandersson and Elias Sonnsjö Lönegren. "Neural Networks for modelling of a virtual sensor in an engine". MA thesis. Chalemrs University of Technology, 2019.

[3] Sven Apel and Christian Kästner. "An Overview of Feature-Oriented SoftwareDevelopment". In: *Jorunal of Object Technology* 8.5 (July 2009), pp. 49–84.

[4] Timothy Beasley, Stephen Erickson, and David Allison. "Rank-Based Inverse Normal Transformations are Increasingly Used, But are They Merited?" In: *Behavior Genetics* 39 (Sept. 2009), pp. 580–595. DOI: 10.1007/s10519-009-9281-0.

[5] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1045-9227. DOI: 10.1109/72.279181.

[6] Purnima Bholowalia and Arvind Kumar. "EBK-Means: A Clustering Technique based on Elbow Method and K-Means in WSN". In: *International Journal of Computer Applications* 105.9 (Nov. 2014), pp. 17–24.

[7] Partha Bishnu and Vandana Bhattacharjee. "Software Fault Prediction Using Quad Tree-Based K-Means Clustering Algorithm". In: *Knowledge and Data Engineering, IEEE Transactions on* 24 (June 2012), pp. 1146–1150. DOI: 10.1109/TKDE.2011.163.

[8] Anastasia Borovykh, Sander Bohte, and Cornelis W Oosterlee. "Conditional time series forecasting with convolutional neural networks". In: *arXiv preprint arXiv:1703.04691* (2017).

[9] Encyclopedia Britannica. *NP-complete problem*. 2019. URL: https://www.britannica.com/science/NP-complete-problem (visited on 09/29/2019).

[10] Michelle Cartwright and Martin Shepperd. "An empirical investigation of an object-oriented software system". In: *IEEE Transactions on software engineering* 26.8 (2000), pp. 786–796.

[11] Delphine S. Courvoisier et al. "Performance of logistic regression modeling: beyond the number of events per variable, the role of data structure". In:

Journal of Clinical Epidemiology 64.9 (2011), pp. 993–1000. ISSN: 0895-4356. DOI: https://doi.org/10.1016/j.jclinepi.2010.11.012. URL: http://www.sciencedirect.com/science/article/pii/S0895435610004245.

[12] Arden Dertat. *Applied Deep Learning - Part 4: Convolutional Neural Networks*. 2017. URL: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2 (visited on 09/29/2019).

[13] Sander Dielman. *Classifying plankton with deep neural nets*. 2015. URL: http://benanne.github.io/2015/03/17/plankton.htm (visited on 08/25/2019).

[14] Subhajit Dutta and Marc G. Genton. "Depth-weighted robust multivariate regression with application to sparse data". In: *Canadian Journal of Statistics* 45.2 (2017), pp. 164–184. DOI: 10.1002/cjs.11315. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cjs.11315. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cjs.11315.

[15] Norman E. Fenton and Martin Neil. "A Critique of Software Defect Prediction Models". In: *IEEE Trans. Software Eng.* 25 (1999), pp. 675–689.

[16] Martha Dais Ferreira et al. "Designing architectures of convolutional neural networks to solve practical problems". In: *Expert Systems with Applications* 94 (2018), pp. 205–217. ISSN: 0957-4174. URL: https://doi.org/10.1016/j.eswa.2017.10.052..

[17] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow. Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Incorporated, 2017. ISBN: 9781491962299.

[18] Robert L. Glass. "Frequently Forgotten Fundamental Facts about Software Engineering". In: *IEEE Software* 18.3 (May 2001), pp. 111–112. ISSN: 1937-4194. DOI: 10.1109/MS.2001.922739.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[20] Lan Guo et al. "Robust prediction of fault-proneness by random forests". In: *15th international symposium on software reliability engineering*. IEEE. 2004, pp. 417–428.

[21] Brian Hambling. *1.6.5 The Pesticide Paradox*. 2010. URL: https://app.knovel.com/hotlink/khtml/id:kt00C454B2/software-testing-an-istqb/pesticide-paradox.

[22] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning : data mining, inference, and prediction.* Springer series in statistics. Springer, 2009. ISBN: 978-0-387-84857-0. URL: http://proxy.lib.chalmers.se/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=cat06296a&AN=clc.b2480748&site=eds-live&scope=site.

[23] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York, NY: Springer New York, 2009. ISBN: 978-0-387-84858-7. DOI: 10.1007/978-0-387-84858-7_2. URL: https://doi.org/10.1007/978-0-387-84858-7_2.

[24] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[25] M M Manjurul Islam et al. “Reliable bearing fault diagnosis using Bayesian inference-based multi-class support vector machines”. In: *The Journal of the Acoustical Society of America* 141 (Feb. 2017), EL89. DOI: 10.1121/1.4976038.

[26] Artem Khurshudov. *Suddenly, a leopard print sofa appears.* 2015. URL: http://rocknrollnerd.github.io/ml/2015/05/27/leopard-sofa.html (visited on 08/24/2019).

[27] Stefan Kramer et al. “Prediction of Ordinal Classes Using Regression Trees”. In: *Fundam. Inform.* 47 (2000), pp. 1–13.

[28] Lov Kumar, Sanjay Misra, and Santanu Ku. Rath. “An empirical analysis of the effectiveness of software metrics and fault prediction model for identifying faulty classes”. In: *Computer Standards Interfaces* 53 (2017), pp. 1–32. ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2017.02.003. URL: http://www.sciencedirect.com/science/article/pii/S0920548916300885.

[29] Libo Li, Stefan Lessmann, and Bart Baesens. “Evaluating Software Defect Prediction Performance: An Updated Benchmarking Study”. In: *SSRN Electronic Journal* (Jan. 2019). DOI: 10.2139/ssrn.3312070.

[30] Gilles Louppe. “Understanding Random Forests: From Theory to Practice”. In: *arXiv e-prints*, arXiv:1407.7502 (July 2014), arXiv:1407.7502. arXiv: 1407.7502 [stat.ML].

[31] T.J. MacCabe et al. *Structured testing.* Tutorial Texts Series. IEEE Computer Society Press, 1983. URL: https://books.google.se/books?id=vtNWAAAAMAAJ.

[32] Ruchika Malhotra. “A systematic review of machine learning techniques for software fault prediction”. In: *Applied Soft Computing* 27 (2015), pp. 504–518. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2014.11.023. URL: http://www.sciencedirect.com/science/article/pii/S1568494614005857.

[33] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521865719, 9780521865715.

[34] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

[35] B. Mehlig. “Artificial Neural Networks”. In: *CoRR* abs/1901.05639 (2019). arXiv: 1901.05639. URL: http://arxiv.org/abs/1901.05639.

[36] Tim Menzies et al. “Metrics that matter”. In: *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.* IEEE. 2002, pp. 51–57.

[37] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning.* The MIT Press, 2012. ISBN: 026201825X, 9780262018258.

[38] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction". In: *Proceedings of the 30th international conference on Software engineering*. ACM. 2008, pp. 181–190.

[39] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. "Predicting the location and number of faults in large software systems". In: *IEEE Transactions on Software Engineering* 31.4 (Apr. 2005), pp. 340–355. DOI: 10.1109/TSE.2005.49.

[40] Danijel Radjenović et al. "Software fault prediction metrics: A systematic literature review". In: *Information and Software Technology* 55.8 (2013), pp. 1397–1418. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.02.009. URL: http://www.sciencedirect.com/science/article/pii/S0950584913000426.

[41] R.V. Raghav, G. Lemaitre, and T. Unterthiner. *Compare the effect of different scalers on data with outliers*. 2019. URL: https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html (visited on 08/14/2019).

[42] Rakesh Rana et al. "Analyzing defect inflow distribution and applying Bayesian inference method for software defect prediction in large software projects". In: *Journal of Systems and Software* 117 (2016), pp. 229–244. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.02.015. URL: http://www.sciencedirect.com/science/article/pii/S0164121216000480.

[43] Rakesh Rana et al. "Defect Prediction over Software Life Cycle in Automotive Domain State of the Art and Road Map for Future". In: Aug. 2014. DOI: 10.5220/0005099203770382.

[44] Peter J. Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis". In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: http://www.sciencedirect.com/science/article/pii/0377042787901257.

[45] Stuart Russell and Peter Norvig. *Artificial Intelligence: Pearson New International Edition*. New York, NY: Pearson Education M.U.A., 2013. ISBN: 9781292037172.

[46] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning and Data Mining*. 2nd. Springer Publishing Company, Incorporated, 2017. ISBN: 9781489976857.

[47] Rakhi Sasidharan and Padmamala Sriram. "Hyper-Quadtree-Based K-Means Algorithm for Software Fault Prediction". In: *Computational Intelligence, Cyber Security and Computational Models*. Ed. by G. Sai Sundara Krishnan et al. New Delhi: Springer India, 2014, pp. 107–118. ISBN: 978-81-322-1680-3.

[48] SciKit-Learn. *sklearn.metrics.silhouette$_s$core*. 2019. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html (visited on 09/29/2019).

[49] SciKit-learn. *MinMax Scaler*. 2019. URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html (visited on 08/14/2019).

[50] SciKit-learn. *Quantile Transformer*. 2019. URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.QuantileTransformer.html (visited on 08/15/2019).

[51] SciKit-learn. *sklearn.svm.SVC*. 2019. URL: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC (visited on 09/15/2019).

[52] Zubair Shah et al. "Software Clustering Using Automated Feature Subset Selection". In: vol. 8347. Dec. 2013. DOI: 10.1007/978-3-642-53917-6_5.

[53] Raed Shatnawi and Wei Li. "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process". In: *Journal of Systems and Software* 81.11 (2008), pp. 1868–1882. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2007.12.794. URL: http://www.sciencedirect.com/science/article/pii/S0164121208000095.

[54] Ajmer Singh, Rajesh Bhatia, and Anita Singhrova. "Taxonomy of machine learning algorithms in software fault prediction using object oriented metrics". In: *Procedia Computer Science* 132 (2018). International Conference on Computational Intelligence and Data Science, pp. 993–1001. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.05.115. URL: http://www.sciencedirect.com/science/article/pii/S1877050918308470.

[55] Ajmer Singh, Rajesh Bhatia, and Anita Singhrova. "Taxonomy of machine learning algorithms in software fault prediction using object oriented metrics". In: *Procedia Computer Science* 132 (2018). International Conference on Computational Intelligence and Data Science, pp. 993–1001. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.05.115. URL: http://www.sciencedirect.com/science/article/pii/S1877050918308470.

[56] *Software Quality Assurance - From theory to implemenatation*. Pearsson, Edinburgh Gate, England, 2004.

[57] Chintakindi Srinivas and Chakunta Rao. "A Feature Vector Based Approach for Software Component Clustering and Reuse Using K-means". In: Sept. 2015, pp. 1–5. DOI: 10.1145/2832987.2833080.

[58] Miroslaw Staron and Wilhelm Meding. "Predicting weekly defect inflow in large software projects based on project planning and test status". In: *Information and Software Technology* 50.7 (2008), pp. 782–796. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2007.10.001. URL: http://www.sciencedirect.com/science/article/pii/S0950584907001085.

[59] Miroslaw Staron and Wilhelm Meding. *Software Development Measurement Programs : Development, Management and Evolution*. Springer, 2018. ISBN: 9783319918358. URL: http://search.ebscohost.com/login.aspx?direct=true&AuthType=sso&db=edsebk&AN=1850351&site=eds-live&scope=site&custid=s3911979&authtype=sso&group=main&profile=eds.

[60] Miroslaw Staron, Wilhelm Meding, and Bo Söderqvist. "A method for forecasting defect backlog in large streamline software development projects and its industrial evaluation". In: *Information and Software Technology* 52.10 (2010), pp. 1069–1079. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2010.05.005. URL: http://www.sciencedirect.com/science/article/pii/S0950584910000832.

[61] Dhiauddin Suffian and Suhaimi Ibrahim. "A Prediction Model for System Testing Defects using Regression Analysis". In: *International Journal of Soft Computing and Software Engineering* 2 (Jan. 2014). DOI: 10.7321/jscse.v2.n7.6.

[62] Robert Tibshirani. "Regression shrinkage and selection via the lasso: a retrospective." In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 73.3 (2011), pp. 273–282. ISSN: 13697412. URL: http://proxy.lib.chalmers.se/login?url=http://search.ebscohost.com.proxy.lib.chalmers.se/login.aspx?direct=true&db=buh&AN=60109526&site=ehost-live&scope=site.

[63] Roelf J. Wieringa. *Design science methodology for information systems and software engineering*. Undefined. 10.1007/978-3-662-43839-8. Springer, 2014. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8.

[64] H. Zhang. "An investigation of the relationships between lines of code and defects". In: *2009 IEEE International Conference on Software Maintenance*. Sept. 2009, pp. 274–283. DOI: 10.1109/ICSM.2009.5306304.

[65] Shi Zhong, Taghi Khoshgoftaar, and Naeem Seliya. "Analyzing Software Measurement Data with Clustering Techniques". In: *Intelligent Systems, IEEE* 19 (Apr. 2004), pp. 20–27. DOI: 10.1109/MIS.2004.1274907.